ALP User Documentation

Generated by Doxygen 1.9.3

Chapter 1

ALP User Documentation

The Algebraic Programming (ALP) project is a modern and humble C++ programming framework that achieves scalable and high performance.

With ALP, programmers are encouraged to express programs using algebraic concepts directly. ALP is a humble programming model in that it hides all optimisations pertaining to parallelisation, vectorisation, and other complexities with programming large-scale and heterogeneous systems.

ALP presently exposes the following interfaces:

1. generalised sparse linear algebra, ALP/GraphBLAS;
2. vertex-centric programming, ALP/Pregel.

Several other programming interfaces are under design at present.

For authors who contributed to ALP, please see the NOTICE file.

Contact:

- https://github.com/Algebraic-Programming/ALP
- https://gitee.com/CSL-ALP/graphblas/
- albertjan.yzelman@huawei.com

Author

A. N. Yzelman, Huawei Technologies France (2016-2020)

A. N. Yzelman, Huawei Technologies Switzerland AG (2020-current)

Chapter 2

Deprecated List

Member _GRB_NO_EXCEPTIONS

Support for this macro is being phased out.

Member grb::eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Member grb::eWiseAdd (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Member grb::eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Member grb::eWiseAdd (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Member grb::eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Member grb::eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Member grb::eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Member grb::eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Member grb::finalize ()

Please use grb::Launcher instead. This primitive will be removed from verson 1.0 onwards.

Member grb::foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Vector< MaskType, backend, Coords > &mask, const OP &op=OP(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< MaskType >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

This signature is deprecated. It was implemented for reference (and reference_omp), but could not be implemented for BSP1D and other distributed-memory backends. This signature may be removed with any release beyond 0.6.

Member grb::init (const size_t s, const size_t P, void *const implementation_data)

Please use grb::Launcher instead. This primitive will be removed from verson 1.0 onwards.

Member grb::init ()

Please use grb::Launcher instead. This primitive will be removed from verson 1.0 onwards.

Member grb::OVERLAP

This error code will be replaced with ILLEGAL.

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

Namespace Index

4.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 7

Module Documentation

7.1 ALP/GraphBLAS

ALP/GraphBLAS enables sparse linear algebraic programming.

Modules

- Data Ingestion and Extraction

  Provides functions for putting user data into opaque ALP/GraphBLAS containers, provides functions for extracting data from such containers, and provides query as well resizing functionalities.
- Level-0 Primitives

  A collection of functions that let GraphBLAS operators work on zero-dimensional containers, i.e., on scalars.
- Level-1 Primitives

  A collection of functions that allow ALP/GraphBLAS operators, monoids, and semirings work on a mix of zero-dimensional and one-dimensional containers; i.e., allows various linear algebra operations on scalars and objects of type grb::Vector.
- Level-2 Primitives

  A collection of functions that allow GraphBLAS operators, monoids, and semirings work on a mix of zero-dimensional, one-dimensional, and two-dimensional containers.
- Level-3 Primitives

  A collection of functions that allow GraphBLAS semirings to work on one or more two-dimensional sparse containers (i.e, sparse matrices).

7.1.1 Detailed Description

ALP/GraphBLAS enables sparse linear algebraic programming.

API introduction

ALP/GraphBLAS is an ANSI C++11 variant of the C GraphBLAS standard with a few different choices and an emphasis on portability and auto-parallelisation. It exposes only two containers: grb::Vector and grb::Matrix. A template argument controls the type of the values contained within a container.

A container may have between 0 and c values, and each such value has a coordinate. The value c is the *capacity* of a container, and at most equals the *size* of that container. The size of a matrix is the product of its number of rows and its number of columns. Containers with fewer values than their size are considered *sparse*, while those with as many values as their size are considered *dense*. Scalars correspond to the standard C++ plain-old-data types, and, as such, have size, capacity, and number of values equal to one– scalars are always dense.

For matrices, their size can be derived from grb::nrows and grb::ncols, while for vectors their size may be immediately retrieved via grb::size. For both vectors and matrices, their capacity and current number of values may be retrieved via grb::capacity and grb::nnz, respectively. Finally, containers have a unique identifier that may be retrieved via grb::getID. These identifiers are assigned in a deterministic fashion, so that for deterministic programs executed with the same number of processes, the same containers will be assigned the same IDs.

Containers may be populated using grb::set or by using dedicated I/O routines such as grb::buildVectorUnique or grb::buildMatrixUnique. Here, *unique* refers to the collection of values that should be ingested having no duplicate coordinates; i.e., there are no two values that map to the same coordinate. The first argument to either function is the output container, which is followed by an iterator pair that points to a collection of values to be ingested into the output container.

ALP/GraphBLAS supports multiple user processes P. If $P > 1$, there is a difference between grb::SEQUENTIAL and grb::PARALLEL I/O. The default I/O mode is grb::PARALLEL, which may be overridden by supplying grb::SEQUENTIAL as a fourth and final argument to the input routines. In sequential I/O, the iterator pair must point to the exact same collection of input values on each of the P user processes. In the parallel mode, however, each iterator pair points to disjoint value sets at each of the processes, while their union is what is logically ingested into the output container.

Output iteration is done using the standard STL-style iterators. ALP, however, only supports const_iterators on output. Output iterators default to sequential mode also.

Primitives perform algebraic operations on containers while using explicitly supplied algebraic structures. Primitives may be as simple as the element-wise application of a binary operator to two input vectors, generating values in a third output vector ($z = x \odot y$, grb::eWiseApply), or may be as rich as multiplying two matrices together whose result is to be added in-place to a third matrix ($C \leftarrow C + AB$, grb::mxm). The latter is typically deemed richer since it requires a semiring structure rather than a more basic binary operator.

Primitives are grouped according to their classical BLAS levels:

- Level-0 Primitives
- Level-1 Primitives
- Level-2 Primitives
- Level-3 Primitives

The "level-0" primitives operate on scalars, and in terms of arithmetic intensity match those of level-1 primitives– however, since standard BLAS need not define scalar operations this specification groups them separately. All primitives except for grb::set and grb::eWiseApply are *in-place*, meaning that new output values are "added" to any pre-existing contents in output containers. The operator used for addition is derived from the algebraic structure that the primitive is called with.

ALP requires that every primitive is *parallelisable*. Every backend that implements primitive for a specific system furthermore must specify *performance semantics*. Contrary to functional semantics that this reference specifies, performance semantics guarantee certain observable behaviours when it comes to the amount of work, data movement, synchronisation across parallel systems, and/or memory use.

See also

Performance Semantics

Algebraic Structures

ALP/GraphBLAS defines three types of algebra structures, namely, a

1. binary operator such as grb::operators::add (numerical addition),
2. grb::Monoid, and
3. grb::Semiring.

Binary operators are parametrised in two input domains and one output domain, $D_1 \times D_2 \to D_3$. The D_i are given as template arguments to the operator. A grb::Monoid is composed from a binary operator coupled with an identity. For example, the additive monoid is defined as

```
grb::Monoid<
  grb::operators::add< double >,
  grb::identities::zero
>
```

Note that passing a single domain as a template argument to a binary operator is a short-hand for an operator with $D_{\{1,2,3\}}$ equal to the same domain.

Likewise, a grb::Semiring is composed from two monoids, where the first, the so-called additive monoid, furthermore must be commutative. The classic semiring over integers taught in elementary school, for example, reads

```
grb::Semiring<
  grb::operators::add< unsigned int >,
  grb::operators::mul< unsigned int >,
  grb::identities::zero,
  grb::identities::one
>
```

Monoids and semirings must comply with their regular axioms– a type system assists users by checking for incorrect operators acting as additive or multiplicative monoids. Errors are reported *at compile time*, through the use of *algebraic type traits* such as grb::is_associative.

See also

Algebraic Type Traits

Standard operators and identities are found in their respective namespaces, grb::operators and grb::identities, respectively. The ALP monoids and semirings are generalised from their standard mathematical definitions in that they hold multiple domains. The description of grb::Semiring details the underlying mathematical structure that nevertheless can be identified.

ALP/GraphBLAS by example

An example is provided within examples/sp.cpp. It demonstrates usage of this API. We now follow with some code snippets from that example. First, the example dataset:

```
static const char * const vertex_ids[ 5 ] = { "Shenzhen", "Hong Kong", "Santa Clara", "London", "Paris" };
static const double distances[ 10 ] = { 8.628, 8.964, 11.148, .334, 9.606, 9.610, .017, .334, .017, .334 };
static const int price[ 10 ] = { 723, 956, 600, 85, 468, 457, 333, 85, 50, 150 };
static const double timeliness[ 10 ] = { 0.9, 0.7, 0.99, 0.9, 0.9, 0.7, 0.99, 0.7, .99, 0.99 };
static const std::string mode[ 10 ] = { "air", "air", "air", "air", "air", "air", "air", "air", "land",
    "land" };
static const size_t I[ 10 ] = { 3, 4, 2, 3, 3, 4, 1, 4, 1, 4 };
static const size_t J[ 10 ] = { 2, 2, 1, 4, 1, 1, 0, 3, 0, 3 };
```

Matrix creation (5-by-5 matrix, 10 nonzeroes):

```
    grb::Matrix< double > dist( 5, 5 );
    resize( dist, 10 );
```

Vector creation:

```
    grb::Vector< double > x( 5 );
    grb::Vector< double > y( 5 );
```

Matrix assignment:

```
    buildMatrixUnique( dist, &( I[ 0 ] ), &( J[ 0 ] ), distances, 10, SEQUENTIAL );
```

Vector assignment:

```
    grb::set( x, INFINITY );
    grb::setElement( x, 0.0, 4 );
    grb::set( y, x );
```

Example semiring definition:

```
    grb::Semiring< grb::operators::min< double >, grb::operators::add< double >, grb::identities::infinity,
       grb::identities::zero > shortest_path_double;
```

Example semiring use:

```
    grb::vxm( y, x, dist, shortest_path_double );
```

Example function taking arbitrary semirings:

```
template< typename Ring >
grb::Vector< typename Ring::D4 >
shortest_path( const grb::Matrix< typename Ring::D2 > & A, const grb::Vector< typename Ring::D1 > &
       initial_state, const size_t hops = 1, const Ring & ring = Ring() ) {
    const size_t size = grb::size( initial_state );
    grb::Vector< typename Ring::D4 > ret( size );
    grb::Vector< typename Ring::D4 > new_state( size );
    grb::set( ret, initial_state );
    vxm( ret, initial_state, A, ring );
    for( size_t i = 1; i < hops; ++i ) {
        grb::set( new_state, ret );
        vxm( ret, new_state, A, ring );
    }
    return ret;
}
```

Example use of a function taking arbitrary semirings:

```
    grb::Vector< int > trip_prices = shortest_path< shortest_path_ints >( prices, initial_trip_price, 2 );
```

Full example use case:

```
    grb::Matrix< double > T( 5, 5 );
    resize( T, 10 );
    buildMatrixUnique( T, &( I[ 0 ] ), &( J[ 0 ] ), timeliness, 10, SEQUENTIAL );
    grb::Vector< double > initial_timeliness( 5 );
    grb::set( initial_timeliness, 0.0 );
    grb::setElement( initial_timeliness, 1.0, 4 );
    const grb::Vector< double > trip_timeliness2 = shortest_path< mul_max_double >( T, initial_timeliness, 2
       );
    (void)printf( "If we take a maximum of two separate trips, we can go from Paris "
                  "to the following cities timeliness as follows:\n" );
    for( const std::pair< size_t, double > & pair : trip_timeliness2 ) {
        const size_t i = pair.first;
        const double val = pair.second;
        if( val > 0 ) {
            (void)printf( "--> %s with %lf percent probability of arriving on time\n", vertex_ids[ i ], val
       * 100.0 );
        }
    }
```

Author

A. N. Yzelman, Huawei Technologies France (2016-2020)

A. N. Yzelman, Huawei Technologies Switzerland AG (2020-current)

7.2 ALP/Pregel

ALP/Pregel enables vertex-centric programming.

Classes

- struct ConnectedComponents< VertexIDType >

  A vertex-centric Connected Components algorithm.
- struct PageRank< IOType, localConverge >

  A Pregel-style PageRank-like algorithm.
- class Pregel< MatrixEntryType >

  A Pregel run-time instance.
- struct PregelState

  The state of the vertex-center Pregel program that the user may interface with.

Enumerations

- enum SparsificationStrategy { NONE = 0 , ALWAYS , WHEN_REDUCED , WHEN_HALVED }

  The set of sparsification strategies supported by the ALP/Pregel interface.

Variables

- constexpr const SparsificationStrategy **out_sparsify** = NONE

  What sparsification strategy should be applied to the outgoing messages.

7.2.1 Detailed Description

ALP/Pregel enables vertex-centric programming.

API introduction

With vertex-centric programming, graph algorithms are written from the perspective of a vertex within an input graph. Each vertex executes a program on a round-by-round basis, while between rounds all vertex programs pass messages to neighbour vertices using the edges of the input graph. Edges may be directed or undirected; in the former, messages travel from the source vertex to the destination vertex only. Each vertex program sends the same message to all of its neighbours – i.e., it broadcasts a single given message. In ALP/Pregel, incoming messages are furthermore *accumulated* using a grb::Monoid. The accumulation of incoming messages is typically used by the vertex-centric program during the next round it executes.

Pregel programs thus execute on a given graph, and hence constructing a grb::interfaces::Pregel instance requires passing input iterators corresponding to the graph on which ALP/Pregel programs are executed. Such an instance logically corresponds to an execution engine of vertex-centric programs *for a specific graph*. Multiple grb::interfaces::Pregel instances, each potentially built using a different input graph, may exist simultaneously.

ALP/Pregel programs then are executed using grb::interfaces::Pregel::execute. The first template argument to this function is the binary operator of the monoid to be used for accumalating incoming messages, while the second template argument corresponds to its identity– see grb::operators and grb::identities for example operator and identities. The remainder template arguments to grb::interfaces::Pregel::execute are automatically inferred.

The first non-template argument is the vertex-centric program, for example, grb::algorithms::pregel::ConnectedComponents– a vertex-centric program in ALP/GraphBLAS hence is a class where the program is given as a public static function named *program*. This function takes five arguments:

1. the current state of the vertex (read-write),
2. the incoming message (after accumulation, read only),
3. the outgoing message (read-write),
4. the global program parameters (read only), and
5. the Pregel interface state (read only and read-write).

The types of arguments 1-4 are defined by the program, but must be plain old data (POD) types– similar to the requirements of an ALP operator. An example of an ALP/Pregel algorithm that has non-trivial algorithm parameters is grb::algorithms::pregel::PageRank: grb::algorithms::pregel::PageRank::Data.

The type of the 5th argument to grb::interfaces::Pregel::execute is an instance of grb::interfaces::PregelState. Some of the ALP/Pregel state fields are read-only, such as the current round number grb::interfaces::PregelState::round, while others are read-write. Please see the corresponding documentation for what read-only states may be inspected during program execution. Some fields are global (such as again the current round number), while others are specific to the vertex a program is running on (such as grb::interfaces::PregelState::indegree).

Read-write ALP/Pregel state is used for determining termination conditions. There are two associated flags:

1. grb::interfaces::PregelState::active, and
2. grb::interfaces::PregelState::voteToHalt.

Each vertex has its own state of these two flags, with the defaults being `true` for the former and `false` for the latter.

If, by the end of any round, a vertex sets its `active` flag to `false`, that vertex will not participate in any future rounds. For any neighbouring vertices it shall be as though the inactive vertex keeps broadcasting the identity of the given accumulation monoid.

If at the end of any round all vertices are inactive, the program terminates. Similarly, if by the end of a round *all* vertices have the `voteToHalt` flag set to `true`, then that Pregel program terminates as well.

Using vertex-centric algorithms

By convention, ALP/Pregel algorithms allow for a simplified way of executing them that does not require the Pregel algorithm user to pass the right monoid to grb::interfaces::Pregel::execute each time they call one, such as, for example,

- grb::algorithms::pregel::ConnectedComponents::execute, or
- grb::algorithms::pregel::PageRank::execute.

These functions only take the Pregel instance that is to execute the Pregel program, as well as a vector of initial states as mandatory input. As usual, optional parameters indicate the maximum number of rounds allotted to the program (zero for unbounded), and where to write back the number of rounds after which the program has terminated (`NULL` for no write back).

All pre-defined ALP/Pregel algorithms reside in the grb::algorithms::pregel namespace.

Configuration settings

The ALP/Pregel run-time system manages state for every vertex in the underlying graph. The execution time of a single round is always proportional to the number of active vertices. Since inactive vertices stay inactive in subsequent rounds, their state could be erased. This has two *potential* benefits:

1. it *may* (depending on the used backend's performance semantics) reduce memory use; and/or
2. it *may* result in faster execution (depending on the used backend's performance semantics).

We may opt to always attempt to *sparsify* state, use some heuristic to determine when to sparsify, or just simply never attempt such sparsification.

This choice is configurable via grb::interfaces::config::out_sparsify; see grb::interfaces::config::SparsificationStrategy for options and more details.

7.2.2 Enumeration Type Documentation

7.2.2.1 SparsificationStrategy

`enum SparsificationStrategy`

The set of sparsification strategies supported by the ALP/Pregel interface.

Enumerator

NONE	No sparsification of internal and user-defined vertex states, beyond that which is necessary to bound the runtime by the number of active vertices.

Enumerator

ALWAYS	Always applies the sparsification procedure on both internal and user-defined vertex states. Does not consider whether the resulting operation would reduce the number of vertex entries. This variant was tested against NONE for out_sparsify, and found to be slower always.

Enumerator

WHEN_REDUCED	Sparsify only when the resulting vector would indeed be sparser. While this sounds like it should be a minimal condition to check for before applying sparsification, this check itself comes at non-trivial overhead for any backend. The performance of this strategy versus ALWAYS hence is a trade-off, one that varies with underlying graphs as well as with the vertex-centric

Enumerator

WHEN_HALVED	Sparsify only when the resulting vector would have half (or less) its current number of nonzeroes. This is a simple heuristic that balances the trade-off of *applying* sparsification by amortising its overhead. The overhead described at WHEN_REDUCED corresponding to determining the gain of sparsification, however, remains the same.

Enumerator

7.3 Algebraic Type Traits

Algebraic type traits allows compile-time reasoning on algebraic structures.

Classes

- struct has_immutable_nonzeroes< T >
  Used to inspect whether a given semiring has immutable nonzeroes under addition.
- struct is_associative< T, typename >
  Used to inspect whether a given operator or monoid is associative.
- struct is_commutative< T, typename >
  Used to inspect whether a given operator or monoid is commutative.
- struct is_container< T >
  Used to inspect whether a given type is an ALP/GraphBLAS container.
- struct is_idempotent< T, typename >
  Used to inspect whether a given operator or monoid is idempotent.
- struct is_monoid< T >
  Used to inspect whether a given type is an ALP monoid.
- struct is_object< T >
  Used to inspect whether a given type is an ALP/GraphBLAS object.
- struct is_operator< T >
  Used to inspect whether a given type is an ALP operator.
- struct is_semiring< T >
  Used to inspect whether a given type is an ALP semiring.

7.3.1 Detailed Description

Algebraic type traits allows compile-time reasoning on algebraic structures.

Under *algebraic type traits*, ALP defines two classes of type traits:

1. classical type traits, akin to, e.g., `std::is_integral`, defined over the ALP-specific algebraic objects such as grb::Semiring, and
2. algebraic type traits that allow for the compile-time introspection of algebraic structures.

Under the first class, the following type traits are defined by ALP:

- grb::is_operator, grb::is_monoid, and grb::is_semiring, but also
- grb::is_container and grb::is_object.

Under the second class, the following type traits are defined by ALP:

- grb::is_associative, grb::is_commutative, grb::is_idempotent, and grb::has_immutable_nonzeroes.

Algebraic type traits are a central concept to ALP; depending on algebraic properties, ALP applies different optimisations. Properties such as associativity furthermore often define whether primitives may be automatically parallelised. Therefore, some primitives only allow algebraic structures with certain properties.

Since algebraic type traits are compile-time, the composition of invalid structures (e.g., composing a monoid out of a non-associative binary operator), or the calling of a primitive using an incompatible algebraic structure, results in an *compile-time* error. Such errors are furthermore accompanied by clear messages and suggestions.

7.4 BSP1D backend configuration

All configuration parameters for the BSP1D and hybrid backends.

Classes

- class IMPLEMENTATION< BSP1D >

  This class collects configuration parameters that are specific to the grb::BSP1D and grb::hybrid backends.

7.4.1 Detailed Description

All configuration parameters for the BSP1D and hybrid backends.

7.5 Backends

ALP code is compiled using a compiler wrapper, which optionally takes a backend parameter as an argument.

Classes

- class Properties< backend >

  Collection of various properties on the given ALP/GraphBLAS backend.

Enumerations

- enum Backend {
  reference , reference_omp , hyperdags , nonblocking ,
  shmem1D , **NUMA1D** , **GENERIC_BSP** , BSP1D ,
  doublyBSP1D , **BSP2D** , **autoBSP** , **optBSP** ,
  hybrid , **hybridSmall** , **hybridMid** , **hybridLarge** ,
  minFootprint , banshee , **banshee_ssr** }

  A collection of all backends.

7.5.1 Detailed Description

ALP code is compiled using a compiler wrapper, which optionally takes a backend parameter as an argument.

The backend selection controls for which use case the code is compiled. Options that are always included are:

1. grb::reference, a single-process, auto-vectorising, sequential backend;
2. grb::reference_omp, a single-process, auto-parallelising, shared-memory parallel backend based on Open↩ MP and the aforementioned vectorising backend;
3. grb::hyperdags, a backend that captures the meta-data of computations while delegating the actual work to the grb::reference backend. At program exit, the grb::hyperdags backend dumps a HyperDAG of the computations performed.

Additionally, the following backends may be enabled by providing their dependences before building ALP:

1. grb::BSP1D, an auto-parallelising, distributed-memory parallel backend based on the Lightweight Parallel Foundations (LPF). This is a multi-process backend and may rely on any single-process backend for process-local computations, which by default is grb::reference. Distributed-memory auto-parallelisation is achieved using a row-wise one-dimensional block-cyclic distributon. Its combination with the grb::reference_omp backend results in a fully hybrid shared- and distributed-memory GraphBLAS implementation.
2. grb::hybrid, essentially the same backend as grb::BSP1D, but now composed with the grb::reference_omp backend for process-local computations. This backend facilitates full hybrid shared- and distributed-memory parallelisation.
3. grb::banshee, a single-process, reference-based backend for the Banshee RISC-V hardware simulator making use of indirection stream semantic registers (ISSR). Written by Dan Iorga in collaboration with ETHZ. This backend is outdated, but, last tested, remained functional.

The grb::Backend enum lists all backends known to ALP. Properties of a backend that may affect more advanced user code are collected in grb::Properties.

Author

A. N. Yzelman, Huawei Technologies Switzerland AG (2020-current)

7.5.2 Enumeration Type Documentation

7.5.2.1 Backend

```
enum Backend
```

A collection of all backends.

Depending on which dependences were configured during the bootstrapping of this ALP installation, some of these backends may be disabled.

Enumerator

reference	The sequential reference implementation. Supports fast operations with both sparse and dense vectors, and employs auto-vectorisation.

Enumerator

reference_omp	The threaded reference implementation. Supports fast operations with both sparse and dense vectors. Employs OpenMP used with a mixture of fork/join and SPMD programming styles.

Enumerator

hyperdags	A backend that automatically extracts hyperDAGs from user computations. It only captures metadata for recording the hyperDAG, and relies on another backend to actually execute the requested computations– by default, this is the reference backend.

Enumerator

nonblocking	The threaded non-blocking implementation. Supports fast operations with both sparse and dense vectors. This backend is currently under development.

Enumerator

BSP1D	A parallel implementation based on a row-wise 1D data distribution, implemented using LPF. This backend manages multiple user processes, manages data distributions of containers between those user processes, and decomposes primitives into local compute phases with intermittent communications. For local compute

Enumerator

hybrid	A composed backend that uses reference_omp within each user process and BSP1D between sockets. This backend is implemented using the BSP1D code, with the process-local backend overridden from reference to reference_omp.
banshee	A variant for Snitch RISC-V cores. It is based on an older reference backend.

7.6 Benchmarking

ALP has a specialised class for benchmarking ALP programs, grb::Benchmarker, which is a variant on the grb::Launcher.

Classes

- class Benchmarker< mode, implementation >

  A class that follows the API of the grb::Launcher, but instead of launching the given ALP program once, it launches it multiple times while benchmarking its execution times.

7.6.1 Detailed Description

ALP has a specialised class for benchmarking ALP programs, grb::Benchmarker, which is a variant on the grb::Launcher.

It codes a particular benchmarking strategy of any given ALP program as described below.

The program is called *inner* times *outer* times. Between every *inner* repetitions there is a one-second sleep that ensures machine variability is taken into account. Several statistics are measured across the *outer* repetitions: the minimum, maximum, average, and the (unbiased) sample standard deviation. By contrast, for the *inner* repetitions, only an average is computed – the function of *inner* repetitions is solely to avoid timing programs that execute in too short a time frame, meaning a time frame that is of a similar order as the time it takes to actually call the system timer functionalities.

Note

As a result, *inner* should always equal *one* when benchmarking any non-trivial ALP program, while for benchmarking ALP kernels on small data *inner* may be taken (much) larger.

In published experiments, *inner* is chosen such that a single outer repetition takes 10 to 100 milliseconds.

7.7 Common configuration settings

Configuration elements contained in this group affect all backends.

Classes

- class BENCHMARKING

  Benchmarking default configuration parameters.
- class CACHE_LINE_SIZE

  Contains information about the target architecture cache line size.
- class IMPLEMENTATION< backend >

  Collects a series of implementation choices corresponding to some given backend.
- class MEMORY

  Memory configuration parameters.
- class SIMD_SIZE

  The SIMD size, in bytes.

Functions

- static constexpr size_t big_memory ()
- static constexpr ALLOC_MODE defaultAllocMode ()

  Defines how private memory regions are allocated.
- static constexpr size_t inner ()
- static constexpr size_t l1_cache_size ()
- static constexpr size_t outer ()
- static constexpr ALLOC_MODE sharedAllocMode ()

  Defines how shared memory regions are allocated.

7.7.1 Detailed Description

Configuration elements contained in this group affect all backends.

7.7.2 Function Documentation

7.7.2.1 big_memory()

```
static constexpr size_t big_memory ( )  [inline], [static], [constexpr]
```

Returns

What is considered a lot of memory, in 2-log of bytes.

7.7.2.2 defaultAllocMode()

```
static constexpr ALLOC_MODE defaultAllocMode ( )  [static], [constexpr]
```

Defines how private memory regions are allocated.

Returns

how a memory region that will not be accessed by threads other than the allocating thread, should be allocated.

7.7.2.3 inner()

```
static constexpr size_t inner ( )  [inline], [static], [constexpr]
```

Returns

The default number of inner repetitions.

7.7.2.4 l1_cache_size()

```
static constexpr size_t l1_cache_size ( )  [inline], [static], [constexpr]
```

Returns

the private L1 data cache size, in bytes.

7.7.2.5 outer()

```
static constexpr size_t outer ( )  [inline], [static], [constexpr]
```

Returns

The default number of outer repetitions.

7.7.2.6 sharedAllocMode()

```
static constexpr ALLOC_MODE sharedAllocMode ( )  [static], [constexpr]
```

Defines how shared memory regions are allocated.

Returns

how a memory region that may be accessed by thread other than the allocating thread, should be allocated.

7.8 Configuration

This module collects all configuration settings.

Modules

- BSP1D backend configuration

  All configuration parameters for the BSP1D and hybrid backends.
- Common configuration settings

  Configuration elements contained in this group affect all backends.
- Nonblocking backend configuration

  All configuration parameters for the grb::nonblocking backend.
- Reference and reference_omp backend configuration

  All configuration parameters for the grb::reference and the grb::reference_omp backends.

Classes

- class BENCHMARKING

  Benchmarking default configuration parameters.
- class CACHE_LINE_SIZE

  Contains information about the target architecture cache line size.
- class IMPLEMENTATION< backend >

  Collects a series of implementation choices corresponding to some given backend.
- class MEMORY

  Memory configuration parameters.
- class SIMD_SIZE

  The SIMD size, in bytes.

Typedefs

- typedef unsigned int ColIndexType

  What data type should be used to store column indices.
- typedef size_t NonzeroIndexType

  What data type should be used to refer to an array containing nonzeroes.
- typedef unsigned int RowIndexType

  What data type should be used to store row indices.
- typedef unsigned int VectorIndexType

  What data type should be used to store vector indices.

7.8.1 Detailed Description

This module collects all configuration settings.

7.8.2 Typedef Documentation

7.8.2.1 ColIndexType

```
typedef unsigned int ColIndexType
```

What data type should be used to store column indices.

Some uses cases may require this to be set to `size_t` – others may do with (much) smaller data types instead.

Note

The data type for indices of general arrays is not configurable. This set of implementations use `size_t` for those.

7.8.2.2 NonzeroIndexType

```
typedef size_t NonzeroIndexType
```

What data type should be used to refer to an array containing nonzeroes.

Some uses cases may require this to be set to `size_t`– others may do with (much) smaller data types instead.

Note

The data type for indices of general arrays is not configurable. This set of implementations use `size_t` for those.

7.8.2.3 RowIndexType

```
typedef unsigned int RowIndexType
```

What data type should be used to store row indices.

Some uses cases may require this to be set to `size_t`– others may do with (much) smaller data types instead.

Note

The data type for indices of general arrays is not configurable. This set of implementations use `size_t` for those.

7.8.2.4 VectorIndexType

```
typedef unsigned int VectorIndexType
```

What data type should be used to store vector indices.

Some uses cases may require this to be set to `size_t`– others may do with (much) smaller data types instead.

Note

The data type for indices of general arrays is not configurable. This set of implementations use `size_t` for those.

7.9 Data Ingestion and Extraction

Provides functions for putting user data into opaque ALP/GraphBLAS containers, provides functions for extracting data from such containers, and provides query as well resizing functionalities.

Classes

- class PinnedVector< IOType, implementation >

  Provides a mechanism to access ALP containers from outside of an ALP context.

Functions

- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename fwd_iterator3 = const InputType * __restrict__, Backend implementation = config::default_backend>
  RC buildMatrixUnique (Matrix< InputType, implementation > &A, fwd_iterator1 I, const fwd_iterator1 I_end, fwd_iterator2 J, const fwd_iterator2 J_end, fwd_iterator3 V, const fwd_iterator3 V_end, const IOMode mode)

  Assigns nonzeroes to the matrix from a coordinate format.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename fwd_iterator3 = const InputType * __restrict__, Backend implementation = config::default_backend>
  RC **buildMatrixUnique** (Matrix< InputType, implementation > &A, fwd_iterator1 I, fwd_iterator2 J, fwd_↩ iterator3 V, const size_t nz, const IOMode mode)

  Alias that transforms a set of pointers and an array length to the buildMatrixUnique variant based on iterators.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename RIT , typename CIT , typename NIT , typename fwd_iterator , Backend implementation = config::default_backend>
  RC buildMatrixUnique (Matrix< InputType, implementation, RIT, CIT, NIT > &A, fwd_iterator start, const fwd_iterator end, const IOMode mode)

  Version of buildMatrixUnique that works by supplying a single iterator (instead of three).
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename RIT , typename CIT , typename NIT , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename length_type = size_t, Backend implementation = config::default_backend>
  RC **buildMatrixUnique** (Matrix< InputType, implementation, RIT, CIT, NIT > &A, fwd_iterator1 I, fwd_↩ iterator2 J, const length_type nz, const IOMode mode)

  Version of the above buildMatrixUnique that handles nullptr value pointers.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator , Backend backend, typename Coords >
  RC buildVector (Vector< InputType, backend, Coords > &x, fwd_iterator start, const fwd_iterator end, const IOMode mode)

  Constructs a dense vector from a container of exactly grb::size(x) elements.
- template<Descriptor descr = descriptors::no_operation, typename InputType , class Merger = operators::right_assign< InputType >, typename fwd_iterator1 , typename fwd_iterator2 , Backend backend, typename Coords >
  RC buildVector (Vector< InputType, backend, Coords > &x, fwd_iterator1 ind_start, const fwd_↩ iterator1 ind_end, fwd_iterator2 val_start, const fwd_iterator2 val_end, const IOMode mode, const Merger &merger=Merger())

  Ingests possibly sparse input from a container to which iterators are provided.
- template<Descriptor descr = descriptors::no_operation, typename InputType , class Merger = operators::right_assign< InputType >, typename fwd_iterator1 , typename fwd_iterator2 , Backend backend, typename Coords >
  RC buildVectorUnique (Vector< InputType, backend, Coords > &x, fwd_iterator1 ind_start, const fwd_↩ iterator1 ind_end, fwd_iterator2 val_start, const fwd_iterator2 val_end, const IOMode mode)

  Ingests a set of nonzeroes into a given vector x.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t capacity (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Queries the capacity of the given ALP/GraphBLAS container.
- template<typename InputType , Backend backend, typename Coords >
  size_t capacity (const Vector< InputType, backend, Coords > &x) noexcept

  Queries the capacity of the given ALP/GraphBLAS container.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  RC clear (Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

Clears a given matrix of all nonzeroes.

- template<typename DataType , Backend backend, typename Coords >
  RC clear (Vector< DataType, backend, Coords > &x) noexcept

  Clears a given vector of all nonzeroes.
- template<typename ElementType , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_backend>
  uintptr_t getID (const Matrix< ElementType, implementation, RIT, CIT, NIT > &x)

  Specialisation of getID for matrix containers.
- template<typename ElementType , typename Coords , Backend implementation = config::default_backend>
  uintptr_t getID (const Vector< ElementType, implementation, Coords > &x)

  Function that returns a unique ID for a given non-empty container.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t ncols (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Requests the column size of a given matrix.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t nnz (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Retrieve the number of nonzeroes contained in this matrix.
- template<typename DataType , Backend backend, typename Coords >
  size_t nnz (const Vector< DataType, backend, Coords > &x) noexcept

  Request the number of nonzeroes in a given vector.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t nrows (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Requests the row size of a given matrix.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  RC resize (Matrix< InputType, backend, RIT, CIT, NIT > &A, const size_t new_nz) noexcept

  Resizes the nonzero capacity of this matrix.
- template<typename InputType , Backend backend, typename Coords >
  RC resize (Vector< InputType, backend, Coords > &x, const size_t new_nz) noexcept

  Resizes the nonzero capacity of this vector.
- template<Descriptor descr = descriptors::no_operation, typename DataType , typename T , typename Coords , Backend backend>
  RC set (Vector< DataType, backend, Coords > &x, const T val, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr) noexcept

  Sets all elements of a vector to the given value.
- template<Descriptor descr = descriptors::no_operation, typename DataType , typename MaskType , typename T , Backend backend, typename Coords >
  RC set (Vector< DataType, reference, Coords > &x, const Vector< MaskType, backend, Coords > &mask, const T val, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr)

  Sets all elements of a vector to the given value whenever the given mask evaluates `true`.
- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType , Backend backend, typename Coords >
  RC set (Vector< OutputType, backend, Coords > &x, const Vector< InputType, backend, Coords > &y, const Phase &phase=EXECUTE)

  Sets the content of a given vector x to be equal to that of another given vector y.
- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename MaskType , typename InputType , Backend backend, typename Coords >
  RC set (Vector< OutputType, backend, Coords > &x, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType, backend, Coords > &y, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType >::value, void >::type *const =nullptr)

  Sets the content of a given vector x to be equal to that of another given vector y.

- template<Descriptor descr = descriptors::no_operation, typename DataType , typename T , Backend backend, typename Coords >
  RC setElement (Vector< DataType, backend, Coords > &x, const T val, const size_t i, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr)

  Sets the element of a given vector at a given position to a given value.
- template<typename DataType , Backend backend, typename Coords >
  size_t size (const Vector< DataType, backend, Coords > &x) noexcept

  Request the size of a given vector.
- template<Backend backend = config::default_backend>
  RC wait ()

  Depending on the backend, ALP/GraphBLAS primitives may be non-blocking, meaning that the operation immediately returns even though the requested computation has not been performed.
- template<Backend backend, typename InputType , typename RIT , typename CIT , typename NIT , typename... Args>
  RC wait (const Matrix< InputType, backend, RIT, CIT, NIT > &A, const Args &... args)

  A variant of grb::wait that executes, at minimum, all nonblocking primitives required for computing a given output matrix as well as, optionally, for any additional output containers given in the variadic argument list.
- template<Backend backend, typename InputType , typename Coords , typename... Args>
  RC wait (const Vector< InputType, backend, Coords > &x, const Args &... args)

  A variant of grb::wait that executes, at minimum, all nonblocking primitives required for computing a given output vector as well as, optionally, for any additional output containers given in the variadic argument list.

7.9.1 Detailed Description

Provides functions for putting user data into opaque ALP/GraphBLAS containers, provides functions for extracting data from such containers, and provides query as well resizing functionalities.

ALP/GraphBLAS operates on opaque data objects. Users can input data using grb::buildVector and/or grb::build↩ Matrix.

The standard output methods are provided by grb::Vector::cbegin and grb::Vector::cend, and similarly for grb::Matrix. Iterators provide parallel output (see IOMode for a discussion on parallel versus sequential IO).

Sometimes it is desired to have direct access to ALP/GraphBLAS memory area, and to have that memory available even after the ALP/GraphBLAS context has been destroyed. This functionality is provided by the concept of *pinned containers* such as provided by grb::PinnedVector.

Containers may be instantiated with default or given requested capacities. Implementations may reserve a higher capacity, but must allocate at least the requested amount or otherwise raise an out-of-memory error.

Capacities are always expressed in terms of number of nonzeroes that the container can hold. Current capacities of container instances can be queried using grb::capacity. At any point in time, the actual number of nonzeroes held within a container is given by grb::nnz and must be less than the reported capacity.

To remove all nonzeroes from a container, see grb::clear. The use of this function does not affect a container's capacity.

Capacities can be resized after a container has been instantiated by use of grb::resize. Smaller capacities may or may not yield a reduction of memory used – this depends on the implementation, and specifically on the memory usage semantics it defines.

After instantiation, the size of a container cannot be modified. The size is retrieved through grb::size for vectors, and through grb::nrows as well as grb::ncols for matrices.

In the above, implementation can also be freely substituted with backend, in that a single implementation can provide multiple backends that define different performance and memory semantics.

7.9.2 Function Documentation

7.9.2.1 buildMatrixUnique() [1/2]

```
RC grb::buildMatrixUnique (
            Matrix< InputType, implementation > & A,
            fwd_iterator1 I,
            const fwd_iterator1 I_end,
            fwd_iterator2 J,
            const fwd_iterator2 J_end,
            fwd_iterator3 V,
            const fwd_iterator3 V_end,
            const IOMode mode )
```

Assigns nonzeroes to the matrix from a coordinate format.

Invalidates any prior existing content. Disallows different nonzeroes to have the same row and column coordinates; input must consist out of unique triples.

Warning

Calling this function with duplicate input coordinates will lead to undefined behaviour.

Template Parameters

descr	The descriptor used. The default is grb::descriptors::no_operation, which means that no pre- or post-processing of
fwd_iterator1	The type of the row index iterator.
fwd_iterator2	The type of the column index iterator.
fwd_iterator3	The type of the nonzero value iterator.
length_type	The type of the number of elements in each iterator.

Parameters

out	*A*	Where to store the given nonzeroes.
in	*I*	A forward iterator to *cap* row indices.

Parameters

in	*J*	A forward iterator to *cap* column indices.
in	*V*	A forward iterator to *cap* nonzero values.
in	*I_end*	A forward iterator in end position relative to *I*.
in	*J_end*	A forward iterator in end position relative to *J*.
in	*V_end*	A forward iterator in end position relative to *V*.

The iterators will only be used to read from, never to assign to.

Parameters

in	*mode*	Whether the input should happen in grb::SEQUENTIAL or in the grb::PARALLEL mode.

In the below, let *nz* denote the number of items pointed to by the iterator pair *I*, *I_end*. This number should match the number of elements in *J*, *J_end and V*, *V_end*.

Returns

grb::SUCCESS When the function completes successfully.

grb::MISMATCH When an element from *I* dereferences to a value larger than the row dimension of this matrix, or when an element from *J* dereferences to a value larger than the column dimension of this matrix. When this error code is returned the state of this container will be as though this function was never called; however, the given forward iterators may have been copied and the copied iterators may have incurred multiple increments and dereferences.

grb::OVERFLW When the internal data type used for storing the number of nonzeroes is not large enough to store the number of nonzeroes the user wants to assign. When this error code is returned the state of this container will be as though this function was never called; however, the given forward iterators may have been copied and the copied iterators may have incurred multiple increments and dereferences.

Warning

This is an expensive function. Use sparingly and only when absolutely necessary.

Note

Streaming input can be implemented by supplying buffered iterators to ALP.

The functionality herein described is exactly that of buildMatrix, though with stricter input requirements. These requirements allow much faster construction.

No masked version of this variant is provided. The use of masks in matrix construction is costly and the user is referred to the costly buildMatrix() function instead.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.2 buildMatrixUnique() [2/2]

```
RC grb::buildMatrixUnique (
            Matrix< InputType, implementation, RIT, CIT, NIT > & A,
            fwd_iterator start,
            const fwd_iterator end,
            const IOMode mode )
```

Version of buildMatrixUnique that works by supplying a single iterator (instead of three).

This is useful in cases where the input is given as a single struct per nonzero, whatever this struct may be exactly, as opposed to multiple containers for row indices, column indices, and nonzero values.

This GraphBLAS implementation provides both input modes since which one is more appropriate (and performant!) depends mostly on how the data happens to be stored in practice.

Template Parameters

descr	The currently active descriptor.
InputType	The value type the output matrix expects.
fwd_iterator	The iterator type.
implementation	For which backend a matrix is being read.

The iterator *fwd_iterator*, in addition to being STL-compatible, must support the following three public functions:

1. `S fwd_iterator.i();` which returns the row index of the current nonzero;
2. `S fwd_iterator.j();` which returns the columnindex of the current nonzero;
3. `V fwd_iterator.v();` which returns the nonzero value of the current nonzero.

It also must provide the following public typedefs:

1. `fwd_iterator::RowIndexType`
2. `fwd_iterator::ColumnIndexType`
3. `fwd_iterator::ValueType`

This means a specialised iterator is required for use with this function. See, for example, grb::utils::internal::Matrix↩ FileIterator.

Parameters

`out`	*A*	The matrix to be filled with nonzeroes from *start* to *end*.

Parameters

in	*start*	Iterator pointing to the first nonzero to be added.
in	*end*	Iterator pointing past the last nonzero to be added.
in	*mode*	Whether the input should happen in grb::SEQUENTIAL or in the grb::PARALLEL mode.

7.9.2.3 buildVector() [1/2]

```
RC grb::buildVector (
            Vector< InputType, backend, Coords > & x,
            fwd_iterator start,
            const fwd_iterator end,
            const IOMode mode )
```

Constructs a dense vector from a container of exactly grb::size(x) elements.

This function aliases to the buildVector routine that takes an accumulator, using grb::operators::right_assign (thus overwriting any old contents).

7.9.2.4 buildVector() [2/2]

```
RC grb::buildVector (
            Vector< InputType, backend, Coords > & x,
            fwd_iterator1 ind_start,
            const fwd_iterator1 ind_end,
            fwd_iterator2 val_start,
            const fwd_iterator2 val_end,
```

```
            const IOMode mode,
            const Merger & merger = Merger() )
```

Ingests possibly sparse input from a container to which iterators are provided.

This function dispatches to the buildVector routine that includes an accumulator, here set to grb::operators::right_assign. Any existing values in *x* that overlap with newer values will hence be overwritten.

7.9.2.5 buildVectorUnique()

```
RC grb::buildVectorUnique (
            Vector< InputType, backend, Coords > & x,
            fwd_iterator1 ind_start,
            const fwd_iterator1 ind_end,
            fwd_iterator2 val_start,
            const fwd_iterator2 val_end,
            const IOMode mode )
```

Ingests a set of nonzeroes into a given vector *x*.

Old values will be overwritten. The given set of nonzeroes must not contain duplicate nonzeroes that should be stored at the same index.

Warning

Inputs with duplicate nonzeroes when passed into this function will invoke undefined behaviour.

Parameters

in,out	*x*	The vector where to ingest nonzeroes into.
in	*ind_start*	Start iterator to the nonzero indices.
in	*ind_end*	End iterator to the nonzero indices.
in	*val_start*	Start iterator to the nonzero values.

Parameters

in	*val_end*	End iterator to the nonzero values.
in	*mode*	Whether sequential or parallel ingestion is requested.

The containers the two iterator pairs point to must contain an equal number of elements. Any pre-existing nonzeroes that do not overlap with any nonzero between *ind_start* and *ind_end* will remain unchanged.

Returns

grb::SUCCESS When ingestion has completed successfully.

grb::ILLEGAL When a nonzero has an index larger than grb::size.

grb::PANIC If an unmitigable error has occured during ingestion.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.6 capacity() [1/2]

```
size_t grb::capacity (
            const Matrix< InputType, backend, RIT, CIT, NIT > & A )  [noexcept]
```

Queries the capacity of the given ALP/GraphBLAS container.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

in	*A*	The matrix whose capacity is requested.

A call to this function shall always succeed and shall never throw exceptions.

Performance semantics.

A call to this function:

1. completes in $\Theta(1)$ work.
2. moves $\Theta(1)$ intra-process data.
3. moves 0 inter-process data.
4. does not require inter-process reduction.
5. leaves memory requirements of *A* untouched.
6. does not make system calls, and in particular shall not allocate or free any dynamic memory.

Note

This is a getter function which has strict performance semantics that are *not* backend-specific.

Backends thus are forced to cache current cacacities and immediately return those. By RAII principles, given containers on account of being instantiated, must have a capacity that can be immediately returned.

7.9.2.7 capacity() [2/2]

```
size_t grb::capacity (
            const Vector< InputType, backend, Coords > & x )  [noexcept]
```

Queries the capacity of the given ALP/GraphBLAS container.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

in	*x*	The vector whose capacity is requested.

A call to this function shall always succeed and shall never throw exceptions.

Performance semantics.

A call to this function:

1. completes in $\Theta(1)$ work.
2. moves $\Theta(1)$ intra-process data.
3. moves 0 inter-process data.
4. does not require inter-process reduction.
5. leaves memory requirements of *x* unchanged.
6. does not make system calls, and in particular shall not allocate or free any dynamic memory.

Note

This is a getter function which has strict performance semantics that are *not* backend-specific.

Backends thus are forced to cache current cacacities and immediately return those. By RAII principles, given containers on account of being instantiated, must have a capacity that can be immediately returned.

7.9.2.8 clear() [1/2]

```
RC grb::clear (
            Matrix< InputType, backend, RIT, CIT, NIT > & A )  [noexcept]
```

Clears a given matrix of all nonzeroes.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

in,out	*A*	The matrix of which to remove all nonzero values.

A call to this function shall always succeed and shall never throw exceptions. That clearing a container should never fail is also an implied requirement of the specification of grb::resize.

On function exit, this matrix contains zero nonzeroes. The matrix dimensions (i.e., row and column sizes) as well as the nonzero capacity remains unchanged.

Returns

grb::SUCCESS This function cannot fail.

Warning

Calling clear may not clear any dynamically allocated memory associated with *A*.

Note

Depending on the memory usage semantics defined on a per-backend basis, grb::resize may or may not free dynamically allocated memory associated with *A*.

Only the destruction of *A* would ensure all corresponding memory is freed, for all backends.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.9 clear() [2/2]

```
RC grb::clear (
            Vector< DataType, backend, Coords > & x )  [noexcept]
```

Clears a given vector of all nonzeroes.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

`in,out`	*x*	The vector of which to remove all values.

A call to this function shall always succeed and shall never throw exceptions. That clearing a container should never fail is also an implied requirement of the specification of grb::resize.

On function exit, this vector contains zero nonzeroes. The vector size as well as its nonzero capacity remain unchanged.

Returns

grb::SUCCESS This function cannot fail.

Warning

Calling clear may not free any dynamically allocated memory associated with *x*. None of the present backends in fact do so.

Note

Even grb::resize may or may not free dynamically allocated memory associated with *x*-- depending on the memory usage semantics defined on a per-backend basis, this is optional.

Only the destruction of *x* would ensure all corresponding memory is freed, for all backends.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.10 getID() [1/2]

```
uintptr_t grb::getID (
            const Matrix< ElementType, implementation, RIT, CIT, NIT > & x )
```

Specialisation of getID for matrix containers.

The same specification applies.

See also

getID

7.9.2.11 getID() [2/2]

```
uintptr_t grb::getID (
            const Vector< ElementType, implementation, Coords > & x )
```

Function that returns a unique ID for a given non-empty container.

Note

An empty container is either a vector of size 0 or a matrix with one of its dimensions equal to 0.

The ID is unique across all currently valid container instances. If n is the number of such valid instances, the returned ID may *not* be strictly smaller than n – i.e., implementations are not required to maintain consecutive IDs (nor would this be possible if IDs are to be reused).

The use of `uintptr_t` to represent IDs guarantees that, at any time during execution, there can never be more initialised containers than can be assigned an ID. Therefore this specification demands that a call to this function never fails.

An ID, once given, may never change during the life-time of the given container. I.e., multiple calls to this function using the same argument must return the same ID.

If the program calling this function is deterministic, then it must assign the exact same IDs across different runs.

If the backend supports multiple user processes, the IDs obtained for the same containers but across different processes, may differ. However, across the same run of a deterministic program, the IDs returned within any single user process must, as per the preceding requirement, be the same across different runs that are executed using the same number of user processes.

Parameters

in	*x*	A valid non-empty ALP container to retrieve a unique ID for.

Note

If *x* is invalid or empty then a call to this function results in undefined behaviour.

Returns

The unique ID corresponding to *x*.

Warning

The returned ID is not the same as a pointer to *x*, since, for example, two containers may be swapped via `std::swap`. In such a case, the IDs of the two containers are swapped also.

Note

Another example is when move semantics are invoked, e.g., when a temporary container is copied into another just before it would be destroyed. Via move semantics the remaining container is in fact not a copy of the temporary one, which would have caused their IDs to be different. Instead, the remaining container has taken over the ownership of the to-be destroyed one, retaining its ID.

For the purposes of defining determinism of ALP programs, and perhaps superfluously, two program which only differ by one program constructing a matrix while the other program constructing a vector, are not considered to be the same program; i.e., implementations are allowed to assign vector IDs differently from matrix IDs. However, implementations are not allowed to run out of IDs to assign as a result of using such a mechanism.

7.9.2.12 ncols()

```
size_t grb::ncols (
            const Matrix< InputType, backend, RIT, CIT, NIT > & A )  [noexcept]
```

Requests the column size of a given matrix.

The column size is set at construction of the given matrix and cannot be changed after instantiation.

A call to this function shall always succeed.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

in	*A*	The matrix of which to retrieve the column size.

Returns

The number of columns of *A*.

This function shall not raise exceptions.

Performance semantics.

A call to this function:

1. completes in $\Theta(1)$ work.
2. moves $\Theta(1)$ intra-process data.
3. moves 0 inter-process data.
4. does not require inter-process reduction.
5. leaves memory requirements of *A* unchanged.
6. does not make system calls, and in particular shall not allocate or free any dynamic memory.

Note

This is a getter function which has strict performance semantics that are *not* backend-specific.

This specification forces implementations and backends to cache the column size of a matrix so that it can be immediately returned. By RAII principles, given containers, on account of being instantiated and passed by reference, indeed must have a size that can be immediately returned.

7.9.2.13 nnz() [1/2]

```
size_t grb::nnz (
            const Matrix< InputType, backend, RIT, CIT, NIT > & A )  [noexcept]
```

Retrieve the number of nonzeroes contained in this matrix.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

in	*A*	The matrix whose current number of nonzeroes is requested.

A call to this function shall always succeed and shall never throw exceptions.

Returns

The number of nonzeroes that *A* contains.

Performance semantics.

A call to this function:

1. completes in $\Theta(1)$ work.
2. moves $\Theta(1)$ intra-process data.
3. moves 0 inter-process data.
4. does not require inter-process reduction.
5. leaves memory requirements of *A* untouched.
6. does not make system calls, and in particular shall not allocate or free any dynamic memory.

Note

This is a getter function which has strict performance semantics that are *not* backend-specific.

Backends thus are forced to cache the current number of nonzeroes and immediately return that cached value.

7.9.2.14 nnz() [2/2]

```
size_t grb::nnz (
            const Vector< DataType, backend, Coords > & x )  [noexcept]
```

Request the number of nonzeroes in a given vector.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

in	*x*	The vector whose current number of nonzeroes is requested.

A call to this function shall always succeed and shall never throw exceptions.

Returns

The number of nonzeroes in *x*.

Performance semantics.

A call to this function:

1. completes in $\Theta(1)$ work.
2. moves $\Theta(1)$ intra-process data.
3. moves 0 inter-process data.
4. does not require inter-process reduction.
5. leaves memory requirements of *A* untouched.
6. does not make system calls, and in particular shall not allocate or free any dynamic memory.

Note

This is a getter function which has strict performance semantics that are *not* backend-specific.

Backends thus are forced to cache the current number of nonzeroes and immediately return that cached value.

7.9.2.15 nrows()

```
size_t grb::nrows (
            const Matrix< InputType, backend, RIT, CIT, NIT > & A )  [noexcept]
```

Requests the row size of a given matrix.

The row size is set at construction of the given matrix and cannot be changed after instantiation.

A call to this function shall always succeed.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

in	*A*	The matrix of which to retrieve the row size.

Returns

The number of rows of *A*.

This function shall not raise exceptions.

Performance semantics.

A call to this function:

1. completes in $\Theta(1)$ work.
2. moves $\Theta(1)$ intra-process data.
3. moves 0 inter-process data.
4. does not require inter-process reduction.
5. leaves memory requirements of *A* unchanged.
6. does not make system calls, and in particular shall not allocate or free any dynamic memory.

Note

This is a getter function which has strict performance semantics that are *not* backend-specific.

This specification forces implementations and backends to cache the row size of a matrix so that it can be immediately returned. By RAII principles, given containers, on account of being instantiated and passed by reference, indeed must have a size that can be immediately returned.

7.9.2.16 resize() [1/2]

```
RC grb::resize (
            Matrix< InputType, backend, RIT, CIT, NIT > & A,
            const size_t new_nz )  [noexcept]
```

Resizes the nonzero capacity of this matrix.

Any current contents of the matrix are *not* retained.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

out	*A*	The matrix whose capacity is to be resized.
in	*new_nz*	The number of nonzeroes this matrix is to contain. After a successful call, the container will have space for *at least new_nz* nonzeroes.

The requested *new_nz* must be smaller or equal to product of the number of rows and columns.

After a call to this function, the matrix shall not contain any nonzeroes. This is the case even after an unsuccessful call, with the exception for cases where grb::PANIC is returned– see below.

The size of this matrix is fixed. By a call to this function, only the maximum number of nonzeroes that the matrix may contain can be adapted.

If the matrix has size zero, meaning either zero rows or zero columns (or, as the preceding implies, both), then all calls to this function will be equivalent to a call to grb::clear. In particular, any value of *new_nz* shall be ignored, even ones that would normally be considered illegal (which would be any nonzero value in the case of an empty container).

A request for less capacity than currently already may be allocated, may or may not be ignored. A backend

1. must define memory usage semantics that may be proportional to the requested capacity, and therefore must free any memory that the user has deemed unnecessary. However, a backend
2. could define memory usage semantics that are *not* proportional to the requested capacity, and in that case a performant implementation may choose not to free memory that the user has deemed unnecessary.

Note

However, useful implementations will almost surely define storage costs that are proportional to *new_nz*, and in such cases resizing to smaller capacity must indeed free up unused memory.

Returns

ILLEGAL When *new_nz* is larger than admissable and *A* was non-empty. The capacity of *A* remains unchanged while its contents have been cleared.

OUTOFMEM When the required memory memory could not be allocated. The capacity of *A* remains unchanged while its contents have been cleared.

PANIC When allocation fails for any other reason. The matrix *A* as well as ALP/GraphBLAS, enters an undefined state.

SUCCESS If *A* is non-empty and when sufficient capacity for resizing was available. The matrix *A* has obtained the requested (or a larger) capacity. Its previous contents, if any, have been cleared.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

Warning

For useful backends, this function will indeed imply system calls and incur $\Theta(new\_nz)$ work and data movement costs. It is thus to be considered an expensive function, and should be used sparingly and only when absolutely necessary.

7.9.2.17 resize() [2/2]

```
RC grb::resize (
            Vector< InputType, backend, Coords > & x,
            const size_t new_nz )  [noexcept]
```

Resizes the nonzero capacity of this vector.

Any current contents of the vector are *not* retained.

Template Parameters

InputType	The type of elements contained in the matrix *A*.
backend	The backend of the matrix *A*.

Parameters

out	*x*	The vector whose capacity is to be resized.
in	*new_nz*	The number of nonzeroes this vector is to contain. After a successful call, the container has, at minimum, space for *new_nz* nonzeroes.

The requested *new_nz* must be smaller than or equal to the size of *x*.

Even for non-successful calls to this function, the vector after the call shall not contain any nonzeroes; only if grb::PANIC is returned shall the resulting state of *x* be undefined.

The size of this vector is fixed. By a call to this function, only the maximum number of nonzeroes that the vector may contain can be adapted.

If the vector has size zero, all calls to this function will be equivalent to a call to grb::clear. In particular, any value for *new_nz* shall be ignored, even ones that would normally be considered illegal (which would be any nonzero value in the case of an empty container).

A request for less capacity than currently already may be allocated, may or may not be ignored. A backend

1. must define memory usage semantics that may be proportional to the requested capacity, and therefore must free any memory that the user has deemed unnecessary. However, a backend

2. could define memory usage semantics that are *not* proportional to the requested capacity, and in that case a performant implementation may choose not to free memory that the user has deemed unnecessary.

Returns

ILLEGAL When *new_nz* is larger than admissable and *x* was non-empty. The vector *x* is cleared, but its capacity remains unchanged.

OUTOFMEM When the required memory memory could not be allocated. The vector *x* is cleared, but its capacity remains unchanged.

SUCCESS If *x* is empty (i.e., has grb::size zero).

PANIC When allocation fails for any other reason. The vector *x*, as well as ALP/GraphBLAS, enters an undefined state.

SUCCESS If *x* is non-empty and when sufficient capacity for the resize operation was available. The vector *x* has obtained a capacity of at least *new_nz while* all nonzeroes it previously contained, if any, are cleared.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

Warning

For most implementations, this function will imply system calls, as well as $\Theta(new\_nz)$ work and data movement costs. It is thus to be considered an expensive function, and should be used sparingly and only when absolutely necessary.

7.9.2.18 set() [1/4]

```
RC grb::set (
             Vector< DataType, backend, Coords > & x,
             const T val,
             const Phase & phase = EXECUTE,
             const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object<
T >::value, void >::type * const  = nullptr )  [noexcept]
```

Sets all elements of a vector to the given value.

Unmasked variant.

Template Parameters

descr	The descriptor used for this operation.
DataType	The type of each element in the given vector.
T	The type of the given value.
backend	The backend that implements this function.

Accepted descriptors

1. grb::descriptors::no_operation
2. grb::descriptors::no_casting

Parameters

in,out	*x*	The vector of which every element is to be set to equal *val*. On output, the number of elements shall be equal to the size of *x*.
in	*val*	The value to set each element of *x* to.
in	*phase*	Which grb::Phase the operation is requested. Optional; the default is grb::EXECUTE.

In grb::RESIZE mode:

Returns

grb::OUTOFMEM When *x* could not be resized to hold the requested output, and the current capacity was insufficient.

grb::SUCCESS When the capacity of *x* was resized to guarantee the output of this operation can be contained.

In grb::EXECUTE mode:

Returns

grb::FAILED When *x* did not have sufficient capacity. The vector *x* on exit shall be cleared.

grb::SUCCESS When the call completes successfully.

In grb::TRY mode (experimental and may not be supported):

Returns

grb::FAILED When *x* did not have sufficient capacity. The vector *x* on exit will have contents defined as described for grb::TRY.

grb::SUCCESS When the call completes successfully.

When *descr* includes grb::descriptors::no_casting and if *T* does not match *DataType*, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.19 set() [2/4]

```
RC grb::set (
              Vector< DataType, reference, Coords > & x,
              const Vector< MaskType, backend, Coords > & mask,
              const T val,
              const Phase & phase = EXECUTE,
              const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object<
T >::value, void >::type * const  = nullptr )
```

Sets all elements of a vector to the given value whenever the given mask evaluates `true`.

Template Parameters

descr	The descriptor used for this operation.
DataType	The type of each element in the given vector.
T	The type of the given value.
backend	The backend that implements this function.

Accepted descriptors

1. grb::descriptors::no_operation

2. grb::descriptors::no_casting
3. grb::descriptors::invert_mask
4. grb::descriptors::structural_mask

Parameters

`in,out`	*x*	The vector of which elements are to be set to *val*. On output, the number of elements shall depend on *mask*.
`in`	*mask*	The given mask. How the sparsity structure and values are evaluated depends on the given *desc*.
`in`	*val*	The value to set elements of *x* to.

Parameters

in	*phase*	Which grb::Phase the operation is requested. Optional; the default is grb::EXECUTE.

Warning

An empty *mask*, meaning grb::size(*mask*) is zero, shall be interpreted as though no mask argument was given. In particular, any descriptors pertaining to the interpretation of *mask* shall be ignored.

In grb::RESIZE mode:

Returns

grb::OUTOFMEM When *x* could not be resized to hold the requested output, and the current capacity was insufficient.

grb::SUCCESS When the capacity of *x* was resized to guarantee the output of this operation can be contained.

In grb::EXECUTE mode:

Returns

grb::FAILED When *x* did not have sufficient capacity. The vector *x* on exit shall be cleared.

grb::SUCCESS When the call completes successfully.

In grb::TRY mode (experimental and may not be supported):

Returns

grb::FAILED When *x* did not have sufficient capacity. The vector *x* on exit will have contents defined as described for grb::TRY.

grb::SUCCESS When the call completes successfully.

When *descr* includes grb::descriptors::no_casting and if *T* does not match *DataType*, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.20 set() [3/4]

```
RC grb::set (
            Vector< OutputType, backend, Coords > & x,
            const Vector< InputType, backend, Coords > & y,
            const Phase & phase = EXECUTE )
```

Sets the content of a given vector *x* to be equal to that of another given vector *y*.

Template Parameters

descr	The descriptor of the operation.
OutputType	The type of each element in the output vector.
InputType	The type of each element in the input vector.

Parameters

`in,out`	*x*	The vector to be set.
`in`	*y*	The source vector.

The vector *x* may not be the same as *y*.

Parameters

`in`	*phase*	Which grb::Phase the operation is requested. Optional; the default is grb::EXECUTE.

Accepted descriptors

1. grb::descriptors::no_operation
2. grb::descriptors::no_casting

When *descr* includes grb::descriptors::no_casting and if *InputType* does not match *OutputType*, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.21 set() [4/4]

```
RC grb::set (
            Vector< OutputType, backend, Coords > & x,
            const Vector< MaskType, backend, Coords > & mask,
            const Vector< InputType, backend, Coords > & y,
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType >::value, void >::type * const  = nullptr )
```

Sets the content of a given vector *x* to be equal to that of another given vector *y*.

If an entry with index *i* has that the corresponding *mask* entry evaluates `false`, then that entry shall not be copied into *x*.

The vector *x* may not equal *y*.

Template Parameters

descr	The descriptor of the operation. Optional; default value is grb::descriptors::no_operation.
OutputType	The type of each element in the output vector.
MaskType	The type of each element in the mask vector.
InputType	The type of each element in the input vector.

Parameters

`in,out`	*x*	The vector to be set.
`in`	*mask*	The output mask.
`in`	*y*	The source vector. May not equal *y*.
`in`	*phase*	Which grb::Phase the operation is requested. Optional; the default is grb::EXECUTE.

Accepted descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

When *descr* includes grb::descriptors::no_casting and if *InputType* does not match *OutputType*, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.22 setElement()

```
RC grb::setElement (
            Vector< DataType, backend, Coords > & x,
            const T val,
            const size_t i,
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object<
T >::value, void >::type * const  = nullptr )
```

Sets the element of a given vector at a given position to a given value.

If the input vector *x* already has an element x_i, that element is overwritten to the given value *val*. If no such element existed, it is added and set equal to *val*. The number of nonzeroes in *x* may thus be increased by one due to a call to this function.

The parameter *i* may not be greater or equal than the size of *x*.

Template Parameters

descr	The descriptor to be used during evaluation of this function. Optional; the default descriptor is grb::descriptors::no_ope
DataType	The type of the elements of *x*.
T	The type of the value to be set.

Parameters

in,out	*x*	The vector to be modified.
in	*val*	The value x_i should read after function exit.
in	*i*	The index of the element of *x* to set.
in	*phase*	Which grb::Phase the operation is requested. Optional; the default is grb::EXECUTE.

Returns

grb::SUCCESS Upon successful execution of this operation.

grb::MISMATCH If *i* is greater or equal than the dimension of *x*.

Accepted descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense.

When *descr* includes grb::descriptors::no_casting and if *T* does not match *DataType*, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.9.2.23 size()

```
size_t grb::size (
            const Vector< DataType, backend, Coords > & x )  [noexcept]
```

Request the size of a given vector.

The dimension is set at construction of the given vector and cannot be changed after instantiation.

A call to this function shall always succeed.

Template Parameters

DataType	The type of elements contained in the vector *x*.
backend	The backend of the vector *x*.

Parameters

in	*x*	The vector of which to retrieve the size.

Returns

The size of the vector *x*.

This function shall not raise exceptions.

Performance semantics.

A call to this function:

1. completes in $\Theta(1)$ work.
2. moves $\Theta(1)$ intra-process data.
3. moves 0 inter-process data.
4. does not require inter-process reduction.
5. leaves memory requirements of *x* unchanged.
6. does not make system calls, and in particular shall not allocate or free any dynamic memory.

Note

This is a getter function which has strict performance semantics that are *not* backend-specific.

This specification forces implementations and backends to cache the size of a vector so that it can be immediately returned. By RAII principles, given containers, on account of being instantiated and passed by reference, indeed must have a size that can be immediately returned.

7.9.2.24 wait() [1/3]

```
RC grb::wait ( )
```

Depending on the backend, ALP/GraphBLAS primitives may be non-blocking, meaning that the operation immediately returns even though the requested computation has not been performed.

More formally, while run-time checks that result in grb::MISMATCH must be performed immediately even when a primitive is non-blocking, the detection of other error codes (such as for example the illegal use of a sparse vector) may in fact be deferred, as is of course any attempt to actually perform the requested computation.

A sequence of nonblocking calls may be forced to execute by a call to this primitive, at which point any non-success error code that would have normally been returned by a nonblocking call, will instead be returned by this primitive. If all requested nonblocking calls have executed successfully, then a call to this function shall return grb::SUCCESS.

The are several other cases in which the computation of nonblocking primtives is forced:

1. whenever an output iterator of an output container of any of the non- blocking primitives is requested; and
2. whenever an output container of any of the non-blocking primitives is input to an ALP/GraphBLAS primitive that has scalar output (e.g., grb::dot or folds from a vector into a scalar). A backend may specify additional such *trigger points*.

If a trigger point has no grb::RC return type, then any deferred non-SUCCESS error codes shall materialise as thrown C++ exceptions.

The performance semantics of a trigger point correspond to a sum of the performance semantics of each of the nonblocking primitives it executes.

Note

A good nonblocking backend will in fact incur less data movement by, e.g., fusing low arithmetic intensity operations, whenever possible. Hence the summed performance semantics typically correspond to worst-case bounds.

If automated decisions by a nonblocking backend is unacceptable in certain (parts of a) code base, then manual fusion is preferable. ALP/GraphBLAS provides grb::eWiseLambda for this purpose.

Returns

grb::SUCCESS If all queued non-blocking primitives are executed successfully. If not, any error code prescribed by the non-blocking primitives requested may be returned instead.

7.9.2.25 wait() [2/3]

```
RC grb::wait (
            const Matrix< InputType, backend, RIT, CIT, NIT > & A,
            const Args &...  args )
```

A variant of grb::wait that executes, at minimum, all nonblocking primitives required for computing a given output matrix as well as, optionally, for any additional output containers given in the variadic argument list.

Implementations may elect to execute more than strictly required. In particular, a valid implementation of this variant simply calls grb::wait.

Parameters

in	*A*	The output container which, after a call to this function returns, must be fully computed.

More formally, after a call to this function, retrieving an output iterator of *A* no longer requires triggering any corresponding nonblocking primitives.

Parameters

in	*args*	Any additional containers whose output must be fully computed after a call to this function.

Returns

grb::SUCCESS If the queued non-blocking primitives that are executed as part of a call to this function have executed successfully. If not, any error code prescribed by the non-blocking primitives whose execution was attempted may be returned instead.

7.9.2.26 wait() [3/3]

```
RC grb::wait (
            const Vector< InputType, backend, Coords > & x,
            const Args &...  args )
```

A variant of grb::wait that executes, at minimum, all nonblocking primitives required for computing a given output vector as well as, optionally, for any additional output containers given in the variadic argument list.

Implementations may elect to execute more than strictly required. In particular, a valid implementation of this variant simply calls grb::wait.

Parameters

in	*x*	The output container which, after a call to this function returns, must be fully computed.

More formally, after a call to this function, retrieving an output iterator of *x* no longer requires triggering any corresponding nonblocking primitives.

Parameters

in	*args*	Any additional containers whose output must be fully computed after a call to this function.

Returns

grb::SUCCESS If the queued non-blocking primitives that are executed as part of a call to this function have executed successfully. If not, any error code prescribed by the non-blocking primitives whose execution was attempted may be returned instead.

7.10 Level-0 Primitives

A collection of functions that let GraphBLAS operators work on zero-dimensional containers, i.e., on scalars.

Functions

- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType1 , typename InputType2 , typename OutputType >
  static enum RC apply (OutputType &out, const InputType1 &x, const InputType2 &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< OutputType >::value, void >::type *=nullptr)

  Out-of-place application of the operator OP on two data elements.
- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType , typename IOType >
  static RC foldl (IOType &x, const InputType &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Application of the operator OP on two data elements.
- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType , typename IOType >
  static RC foldr (const InputType &x, IOType &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Application of the operator OP on two data elements.

7.10.1 Detailed Description

A collection of functions that let GraphBLAS operators work on zero-dimensional containers, i.e., on scalars.

The GraphBLAS uses opaque data types and defines several standard functions to operate on these data types. Examples types are grb::Vector and grb::Matrix, example functions are grb::dot and grb::vxm.

To input data into an opaque GraphBLAS type, each opaque type defines a member function *build*: grb::Vector::build() and grb::Matrix::build().

To extract data from opaque GraphBLAS types, each opaque type provides *iterators* that may be obtained via the STL standard *begin* and *end* functions:

- grb::Vector::begin or grb::Vector::cbegin
- grb::Vector::end or grb::Vector::cend
- grb::Matrix::begin or grb::Matrix::cbegin
- grb::Matrix::end or grb::Matrix::cend

Some GraphBLAS functions, however, reduce all elements in a GraphBLAS container into a single element of a given type. So for instance, grb::dot on two vectors of type grb::Vector<double> using the regular real semiring grb::Semiring<double> will store its output in a variable of type *double*.

When parametrising GraphBLAS functions in terms of arbitrary Semirings, Monoids, Operators, and object types, it is useful to have a way to apply the same operators on whatever type they make functions like grb::dot produce–that is, we require functions that enable the application of GraphBLAS operators on single elements.

This group of BLAS level 0 functions provides this functionality.

7.10.2 Function Documentation

7.10.2.1 apply()

```
static enum RC grb::apply (
             OutputType & out,
             const InputType1 & x,
             const InputType2 & y,
             const OP & op = OP(),
             const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< OutputType >↩
::value, void >::type *  = nullptr )  [static]
```

Out-of-place application of the operator *OP* on two data elements.

The output data will be output to an existing memory location, overwriting any existing data.

Template Parameters

descr	The descriptor passed to this operator.
OP	The type of the oparator to apply.
InputType1	The left-hand side input argument type.
InputType2	The right-hand side input argument type.
OutputType	The output argument type.

Valid descriptors

1. grb::descriptors::no_operation for default behaviour.
2. grb::descriptors::no_casting when a call to this function should *not* automatically cast input arguments to operator input domain, and *not* automatically cast operator output to the output argument domain.

If *InputType1* does not match the left-hand side input domain of *OP*, or if *InputType2* does not match the right-hand side input domain of *OP*, or if *OutputType* does not match the output domain of *OP* while grb::descriptors::no_casting was set, then the code shall not compile.

Parameters

in	*x*	The left-hand side input data.
in	*y*	The right-hand side input data.
out	*out*	Where to store the result of the operator.

Parameters

in	*op*	The operator to apply (optional).

Note

op is optional when the operator type *OP* is explicitly given. Thus there are two ways of calling this function:

1. `double a, b, c; grb::apply< grb::operators::add<double> >( a, b, c );` , or
2. `double a, b, c; grb::operators::add< double > addition_over_doubles; grb::apply( a, b, c, addition_over_doubles);`

There should be no performance difference between the two ways of calling this function. For compatibility with other GraphBLAS implementations, the latter type of call is preferred.

Returns

grb::SUCCESS A call to this function never fails.

Performance semantics.

1. This call comprises $\Theta(1)$ work. The constant factor depends on the cost of evaluating the operator.
2. This call takes $\mathcal{O}(1)$ memory beyond the memory already used by the application when a call to this function is made.
3. This call incurs at most $\Theta(1)$ memory where the constant factor depends on the storage requirements of the arguments and the temporary storage required for evaluation of this operator.

Warning

The use of stateful operators, or even thus use of stateless operators that are not included in grb::operators, may cause this function to incur performance penalties beyond the worst case sketched above.

See also

foldr for applying an operator in-place (if allowed).

foldl for applying an operator in-place (if allowed).

grb::operators::internal::Operator for a discussion on when foldr and foldl successfully generate in-place code.

7.10.2.2 foldl()

```
static RC grb::foldl (
            IOType & x,
            const InputType & y,
            const OP & op = OP(),
            const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object<
InputType >::value &&!grb::is_object< IOType >::value, void >::type *  = nullptr )  [static]
```

Application of the operator *OP* on two data elements.

The output data will overwrite the left-hand side input element.

In mathematical notation, this function calculates $x \odot y$ and copies the result into *x*.

Template Parameters

descr	The descriptor passed to this operator.
OP	The type of the operator to apply.
IOType	The type of the left-hand side input element, which will be overwritten.
InputType	The type of the right-hand side input element. This element will be accessed read-only.

Valid descriptors

1. grb::descriptors::no_operation for default behaviour.
2. grb::descriptors::no_casting when a call to this function should *not* automatically cast input arguments to operator input domain, and *not* automatically cast operator output to the output argument domain.

If *InputType* does not match the right-hand side input domain (see grb::operators::internal::Operator::D2) corresponding to *OP*, then *x* will be temporarily cached and cast into *D2*. If *IOType* does not match the left-hand side input domain corresponding to *OP*, then *y* will be temporarily cached and cast into *D1*. If *IOType* does not match the output domain corresponding to *OP*, then the result of $x \odot y$ will be temporarily cached before cast to *IOType* and written to *y*.

Parameters

`in,out`	*x*	On function entry↩: the left-hand side input parameter. On function exit: the output of the operator.
`in`	*y*	The right-hand side input parameter.
`in`	*op*	The operator to apply (optional).

Returns

grb::SUCCESS A call to this function never fails.

Performance semantics.

1. This call comprises $\Theta(1)$ work. The constant factor depends on the cost of evaluating the operator.
2. This call will not allocate any new dynamic memory.
3. This call requires at most $sizeof(D_1 + D_2 + D_3)$ bytes of temporary storage, plus any temporary requirements for evaluating *op*.
4. This call incurs at most $sizeof(D_1+D_2+D_3)+sizeof(InputType+2IOType)$ bytes of data movement, plus any data movement requirements for evaluating *op*.

Warning

The use of stateful operators, or even thus use of stateless operators that are not included in grb::operators, may cause this function to incur performance penalties beyond the worst case sketched above.

Note

For the standard stateless operators in grb::operators, there are no additional temporary storage requirements nor any additional data movement requirements than the ones mentioned above.

If *OP* is fold-left capable, the temporary storage and data movement requirements are less than reported above.

See also

foldr for a right-hand in-place version.

apply for an example of how to call this function without explicitly passing *op*.

grb::operators::internal Operator for a discussion on fold-right capable operators and on stateful operators.

7.10.2.3 foldr()

```
static RC grb::foldr (
            const InputType & x,
            IOType & y,
            const OP & op = OP(),
            const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object<
InputType >::value &&!grb::is_object< IOType >::value, void >::type *  = nullptr )  [static]
```

Application of the operator *OP* on two data elements.

The output data will overwrite the right-hand side input element.

In mathematical notation, this function calculates $x \odot y$ and copies the result into *y*.

Template Parameters

descr	The descriptor passed to this operator.
~~*OP*~~	~~The type of the operator to apply.~~
InputType	The type of the left-hand side input element. This element will be accessed read-only.
IOType	The type of the right-hand side input element, which will be overwritten.

Valid descriptors

1. grb::descriptors::no_operation for default behaviour.
2. grb::descriptors::no_casting when a call to this function should *not* automatically cast input arguments to operator input domain, and *not* automatically cast operator output to the output argument domain.

If *InputType* does not match the left-hand side input domain (see grb::operators::internal::Operator::D1) corresponding to *OP*, then *x* will be temporarily cached and cast into *D1*. If *IOType* does not match the right-hand side input domain corresponding to *OP*, then *y* will be temporarily cached and cast into *D2*. If *IOType* does not match the output domain corresponding to *OP*, then the result of $x \odot y$ will be temporarily cached before cast to *IOType* and written to *y*.

Parameters

`in`	*x*	The left-hand side input parameter.
`in,out`	*y*	On function entry↩: the right-hand side input parameter. On function exit: the output of the operator.
`in`	*op*	The operator to apply (optional).

Returns

grb::SUCCESS A call to this function never fails.

Performance semantics.

1. This call comprises $\Theta(1)$ work. The constant factor depends on the cost of evaluating the operator.
2. This call will not allocate any new dynamic memory.

3. This call requires at most $sizeof(D_1 + D_2 + D_3)$ bytes of temporary storage, plus any temporary requirements for evaluating *op*.
4. This call incurs at most $sizeof(D_1+D_2+D_3)+sizeof(InputType+2IOType)$ bytes of data movement, plus any data movement requirements for evaluating *op*.

Warning

The use of stateful operators, or even thus use of stateless operators that are not included in grb::operators, may cause this function to incur performance penalties beyond the worst case sketched above.

Note

For the standard stateless operators in grb::operators, there are no additional temporary storage requirements nor any additional data movement requirements than the ones mentioned above.

If *OP* is fold-right capable, the temporary storage and data movement requirements are less than reported above.

See also

foldl for a left-hand in-place version.

apply for an example of how to call this function without explicitly passing *op*.

grb::operators::internal Operator for a discussion on fold-right capable operators and on stateful operators.

7.11 Level-1 Primitives

A collection of functions that allow ALP/GraphBLAS operators, monoids, and semirings work on a mix of zero-dimensional and one-dimensional containers; i.e., allows various linear algebra operations on scalars and objects of type grb::Vector.

Macros

- #define NO_MASK Vector< bool >(0)

  A standard vector to use for mask parameters.

Functions

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , Backend backend, typename Coords >
  RC dot (IOType &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_↩ if< !grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< IOType >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the dot product, $z+ = (x, y)$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class AddMonoid , class AnyOp , typename OutputType , typename InputType1 , typename InputType2 , enum Backend backend, typename Coords >
  RC dot (OutputType &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const AddMonoid &addMonoid=AddMonoid(), const AnyOp &anyOp=AnyOp(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Add↩ Monoid >::value &&grb::is_operator< AnyOp >::value, void >::type *const =nullptr)

Calculates the dot product, $z+ = (x, y)$, under a given additive monoid and multiplicative operator.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + \beta$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + y$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = x. + \beta$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition of two vectors, $z+ = x. + y$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + \beta$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + y$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = x. + \beta$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >

RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

Calculates the element-wise addition of two vectors, $z+ = x. + y$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Computes $z = \alpha \odot \beta$, out of place, monoid version.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Computes $z = \alpha \odot \beta$, out of place, operator version.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Computes $z = \alpha \odot y$, out of place, monoid version.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Computes $z = \alpha \odot y$, out of place, operator version.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Computes $z = x \odot \beta$, out of place, monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Computes $z = x \odot \beta$, out of place, operator variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Computes $z = x \odot y$, out of place, monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = x \odot y$, out of place, operator variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot \beta$, out of place, masked monoid version.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot \beta$, out of place, operator and masked version.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot y$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot y$, out of place, masked operator version.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = x \odot \beta$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Computes $z = x \odot \beta$, out of place, masked operator variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = x \odot y$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = x \odot y$, out of place, masked operator variant.

- template<typename Func , typename DataType , Backend backend, typename Coords , typename... Args>
  RC eWiseLambda (const Func f, const Vector< DataType, backend, Coords > &x, Args...)

  Executes an arbitrary element-wise user-defined function f on any number of vectors of equal length.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of two scalars, $z+ = \alpha . * \beta$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a scalar and vector, $z+ = \alpha . * y$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a vector and scalar, $z+ = x . * \beta$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of two vectors, $z+ = x . * y$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value

&&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

In-place element-wise multiplication of two scalars, $z+ = \alpha . * \beta$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

In-place element-wise multiplication of a scalar and vector, $z+ = \alpha . * y$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

In-place element-wise multiplication of a vector and scalar, $z+ = x . * \beta$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

In-place element-wise multiplication of two vectors, $z+ = x . * y$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , typename IOType , typename InputType , Backend backend, typename Coords >
RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Folds a vector into a scalar, left-to-right.

- template<Descriptor descr = descriptors::no_operation, class Monoid , typename InputType , typename IOType , typename MaskType , Backend backend, typename Coords >
RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Vector< MaskType, backend, Coords > &mask, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< InputType >::value &&!grb::is_object< MaskType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Reduces, or folds, a vector into a scalar.

- template<Descriptor descr = descriptors::no_operation, class OP , typename IOType , typename InputType , typename MaskType , Backend backend, typename Coords >
RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Vector< MaskType, backend, Coords > &mask, const OP &op=OP(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< MaskType >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Folds a vector into a scalar, left-to-right.

- template<Descriptor descr = descriptors::no_operation, class Monoid , typename InputType , typename IOType , typename MaskType , Backend backend, typename Coords >
RC foldr (const Vector< InputType, backend, Coords > &x, const Vector< MaskType, backend, Coords > &mask, IOType &y, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< InputType >::value &&!grb::is_object< MaskType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Folds a vector into a scalar, right-to-left.

- template<Descriptor descr = descriptors::no_operation, class Monoid , typename IOType , typename InputType , Backend backend, typename Coords >
RC foldr (const Vector< InputType, backend, Coords > &y, IOType &x, const Monoid &monoid=Monoid(),

const typename std::enable_if< !grb::is_object< IOType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Folds a vector into a scalar, right-to-left.

7.11.1 Detailed Description

A collection of functions that allow ALP/GraphBLAS operators, monoids, and semirings work on a mix of zero-dimensional and one-dimensional containers; i.e., allows various linear algebra operations on scalars and objects of type grb::Vector.

All functions return an error code of the enum-type grb::RC.

Primitives which produce vector output:

1. grb::set (three variants);
2. grb::foldr (in-place reduction to the right, scalar-to-vector and vector-to-vector);
3. grb::foldl (in-place reduction to the left, scalar-to-vector and vector-to-vector);
4. grb::eWiseApply (out-of-place application of a binary function);
5. grb::eWiseAdd (in-place addition of two vectors, a vector and a scalar, into a vector); and
6. grb::eWiseMul (in-place multiplication of two vectors, a vector and a scalar, into a vector).

Note

When grb::eWiseAdd or grb::eWiseMul using two input scalars is required, consider forming first the resulting scalar using level-0 primitives, and then using grb::set, grb::foldl, or grb::foldr, as appropriate.

Primitives that produce scalar output:

1. grb::foldr (reduction to the right, vector-to-scalar);
2. grb::foldl (reduction to the left, vector-to-scalar).

Primitives that do not require an operator, monoid, or semiring:

1. grb::set (three variants).

Primitives that could take an operator (see grb::operators):

1. grb::foldr, grb::foldl, and grb::eWiseApply. Such operators typically can only be applied on *dense* vectors, i.e., vectors with grb::nnz equal to its grb::size. Operations on sparse vectors require an intepretation of missing vector elements, which monoids or semirings provide.

Therefore, all aforementioned functions are also defined for monoids instead of operators.

The following functions are defined for monoids and semirings, but not for operators alone:

1. grb::eWiseAdd (in-place addition).

The following functions require a semiring, and are not defined for operators or monoids alone:

1. grb::dot (in-place reduction of two vectors into a scalar); and
2. grb::eWiseMul (in-place multiplication).

Sometimes, operations that are defined for semirings we would sometimes also like enabled on *improper* semirings. ALP/GraphBLAS statically checks most properties required for composing proper semirings, and as such, attempts to compose improper ones will result in a compilation error. In such cases, we allow to pass an additive monoid and a multiplicative operator instead of a semiring. The following functions allow this:

1. grb::dot, grb::eWiseAdd, grb::eWiseMul. The given multiplicative operator can be any binary operator, and in particular does not need to be associative.

The algebraic structures lost with improper semirings typically correspond to distributivity, zero being an annihilator to multiplication, as well as the concept of *one*. Due to the latter lost structure, the above functions on impure semirings are *not* defined for pattern inputs.

Warning

I.e., any attempt to use containers of the form
```
grb::Vector<void>
grb::Matrix<void>
```
with an improper semiring will result in a compile-time error.

Note

Pattern containers are perfectly fine to use with proper semirings.

Warning

If an improper semiring does not have the property that the zero identity acts as an annihilator over the multiplicative operator, then the result of grb::eWiseMul may be unintuitive. Please take great care in the use of improper semrings.

For fusing multiple BLAS-1 style operations on any number of inputs and outputs, users can pass their own operator function to be executed for every index *i*.

1. grb::eWiseLambda. This requires manual application of operators, monoids, and/or semirings via level-0 interface – see grb::apply, grb::foldl, and grb::foldr.

For all of these functions, the element types of input and output types do not have to match the domains of the given operator, monoid, or semiring unless the grb::descriptors::no_casting descriptor was passed.

An implementation, whether blocking or non-blocking, should have clear performance semantics for every sequence of graphBLAS calls, no matter whether those are made from sequential or parallel contexts. Backends may define different performance semantics depending on which grb::Phase primitives execute in.

7.11.2 Macro Definition Documentation

7.11.2.1 NO_MASK

```
#define NO_MASK Vector< bool >( 0 )
```

A standard vector to use for mask parameters.

Indicates that no mask shall be used.

7.11.3 Function Documentation

7.11.3.1 dot() [1/2]

```
RC grb::dot (
            IOType & z,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< InputType1 >::value &&!grb::is_object<
InputType2 >::value &&!grb::is_object< IOType >::value &&grb::is_semiring< Ring >::value,
void >::type * const  = nullptr )
```

Calculates the dot product, $z+ = (x, y)$, under a given semiring.

Template Parameters

descr	The descriptor to be used. Optional; default descriptor is grb::descriptors::no_operation.
Ring	The semiring type to use.
OutputType	The output type.
InputType1	The input element type of the left-hand input vector.
InputType2	The input element type of the right-hand input vector.

Parameters

in,out	*z*	The output element $z+ = (x, y)$.
in	*x*	The left-hand input vector *x*.

Parameters

in	*y*	The right-hand input vector *y*.
in	*ring*	The semiring under which to compute the dot product (x, y). The additive monoid is used to accumulate the dot product result into *z*.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH If the dimensions of *x* and *y* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

Valid descriptors

- grb::descriptors::no_operation
- grb::descriptors::no_casting
- grb::descriptors::dense

If the dense descriptor is set, this implementation returns grb::ILLEGAL if it was detected that either *x* or *y* was sparse. In this case, it shall otherwise be as though the call to this function had not occurred (no side effects).

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.2 dot() [2/2]

```
RC grb::dot (
              OutputType & z,
              const Vector< InputType1, backend, Coords > & x,
              const Vector< InputType2, backend, Coords > & y,
              const AddMonoid & addMonoid = AddMonoid(),
              const AnyOp & anyOp = AnyOp(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< AddMonoid >↩
::value &&grb::is_operator< AnyOp >::value, void >::type * const  = nullptr )
```

Calculates the dot product, $z+ = (x, y)$, under a given additive monoid and multiplicative operator.

Template Parameters

descr	The descriptor to be used. Optional; the default descriptors is grb::descriptors::no_operation.
AddMonoid	The monoid used for addition during the computation of (x, y). The same monoid is used for accumulating the result
AnyOp	A binary operator that acts as the multiplication during (x, y).
OutputType	The output type.
InputType1	The input element type of the left-hand input vector.
InputType2	The input element type of the right-hand input vector.

Parameters

in,out	*z*	Where to fold (x, y) into.

Parameters

in	*x*	The left-hand input vector.
in	*y*	The right-hand input vector.
in	*addMonoid*	The additive monoid under which the reduction of the results of element-wise multiplications of *x* and *y* are performed.
in	*anyOp*	The multiplicative operator using which element-wise multiplications of *x* and *y* are performed. This may be any binary operator.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Note

By this primitive by which a dot-product operates under any additive monoid and any binary operator, it follows that a dot product under any semiring can be reduced to a call to this primitive instead.

Returns

grb::MISMATCH When the dimensions of *x* and *y* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::SUCCESS On successful completion of this call.

Valid descriptors

1. grb::descriptors::no_operation
2. grb::descriptors::no_casting
3. grb::descriptors::dense

If the dense descriptor is set, this implementation returns grb::ILLEGAL if it was detected that either *x* or *y* was sparse. In this case, it shall otherwise be as though the call to this function had not occurred (no side effects).

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.3 eWiseAdd() [1/8]

```
RC grb::eWiseAdd (
            Vector< OutputType, backend, Coords > & z,
            const InputType1 alpha,
            const InputType2 beta,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

Calculates the element-wise addition, $z+ = \alpha . + \beta$, under a given semiring.

Note

This is an in-place operation.

Deprecated This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Note

A call to this function is equivalent to two in-place fold operations using the additive monoid of the given semiring. Please update any code that calls grb::eWiseAdd with such a sequence as soon as possible.

We may consider providing this function as an algorithm in the grb::algorithms namespace, similar to grb::algorithms::mpv. Please let the maintainers know if you would prefer such a solution over outright removal and replacement with two folds.

Template Parameters

descr	The descriptor to be used. Optional; default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise addition on.
InputType1	The left-hand side input type to the additive operator of the *ring*.
InputType2	The right-hand side input type to the additive operator of the *ring*.
OutputType	The result type of the additive operator of the *ring*.

Parameters

out	*z*	The output vector of type *OutputType*. This may be a sparse vector.

Parameters

in	*alpha*	The left-hand input scalar of type *Input↩ Type1*.
in	*beta*	The right-hand input scalar of type *Input↩ Type2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Valid descriptors

grb::descriptors::no_operation, grb::descriptors::no_casting, grb::descriptors::dense.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the third domain of *ring* must match *InputType1*, 2) the fourth domain of *ring* must match *InputType2*, 3) the fourth domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.4 eWiseAdd() [2/8]

```
RC grb::eWiseAdd (
            Vector< OutputType, backend, Coords > & z,
            const InputType1 alpha,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

Calculates the element-wise addition, $z+ = \alpha. + y$, under a given semiring.

Note

This is an in-place operation.

Deprecated This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Note

A call to this function is equivalent to two in-place fold operations using the additive monoid of the given semiring. Please update any code that calls grb::eWiseAdd with such a sequence as soon as possible.

We may consider providing this function as an algorithm in the grb::algorithms namespace, similar to grb::algorithms::mpv. Please let the maintainers know if you would prefer such a solution over outright removal and replacement with two folds.

Template Parameters

descr	The descriptor to be used. Optional; default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise addition on.
InputType1	The left-hand side input type to the additive operator of the *ring*.
InputType2	The right-hand side input type to the additive operator of the *ring*.
OutputType	The result type of the additive operator of the *ring*.

Parameters

out	*z*	The output vector of type *OutputType*. This may be a sparse vector.
in	*alpha*	The left-hand input scalar of type *InputType1*.
in	*y*	The right-hand input vector of type *InputType2*. This may be a sparse vector.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *y* and *z* do not match. All input data containers are left untouched; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Valid descriptors

grb::descriptors::no_operation, grb::descriptors::no_casting, grb::descriptors::dense.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the third domain of *ring* must match *InputType1*, 2) the fourth domain of *ring* must match *InputType2*, 3) the fourth domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.5 eWiseAdd() [3/8]

```
RC grb::eWiseAdd (
            Vector< OutputType, backend, Coords > & z,
            const Vector< InputType1, backend, Coords > & x,
            const InputType2 beta,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

Calculates the element-wise addition, $z+ = x. + \beta$, under a given semiring.

Note

This is an in-place operation.

Deprecated This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Note

A call to this function is equivalent to two in-place fold operations using the additive monoid of the given semiring. Please update any code that calls grb::eWiseAdd with such a sequence as soon as possible.

We may consider providing this function as an algorithm in the grb::algorithms namespace, similar to grb::algorithms::mpv. Please let the maintainers know if you would prefer such a solution over outright removal and replacement with two folds.

Template Parameters

descr	The descriptor to be used. Optional; default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise addition on.
InputType1	The left-hand side input type to the additive operator of the *ring*.
InputType2	The right-hand side input type to the additive operator of the *ring*.
OutputType	The result type of the additive operator of the *ring*.

Parameters

out	*z*	The output vector of type *OutputType*. This may be a sparse vector.

Parameters

in	*x*	The left-hand input vector of type *Input↩ Type1*. This may be a sparse vector.
in	*beta*	The right-hand input scalar of type *Input↩ Type2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x* and *z* do not match. All input data containers are left untouched; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Valid descriptors

grb::descriptors::no_operation, grb::descriptors::no_casting, grb::descriptors::dense.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the third domain of *ring* must match *InputType1*, 2) the fourth domain of *ring* must match *InputType2*, 3) the fourth domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.6 eWiseAdd() [4/8]

```
RC grb::eWiseAdd (
            Vector< OutputType, backend, Coords > & z,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

Calculates the element-wise addition of two vectors, $z += x. + y$, under a given semiring.

Note

This is an in-place operation.

Deprecated This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Note

A call to this function is equivalent to two in-place fold operations using the additive monoid of the given semiring. Please update any code that calls grb::eWiseAdd with such a sequence as soon as possible.

We may consider providing this function as an algorithm in the grb::algorithms namespace, similar to grb::algorithms::mpv. Please let the maintainers know if you would prefer such a solution over outright removal and replacement with two folds.

Template Parameters

descr	The descriptor to be used. Optional; default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise addition on.
InputType1	The left-hand side input type to the additive operator of the *ring*.
InputType2	The right-hand side input type to the additive operator of the *ring*.
OutputType	The result type of the additive operator of the *ring*.

Parameters

out	*z*	The output vector of type *OutputType*. This may be a sparse vector.
in	*x*	The left-hand input vector of type *InputType1*. This may be a sparse vector.
in	*y*	The right-hand input vector of type *InputType2*. This may be a sparse vector.

Parameters

in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Note

There is also a masked variant of grb::eWiseAdd, as well as variants where *x* and/or *y* are scalars.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x*, *y*, and *z* do not match. All input data containers are left untouched; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Valid descriptors

grb::descriptors::no_operation, grb::descriptors::no_casting, grb::descriptors::dense.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the third domain of *ring* must match *InputType1*, 2) the fourth domain of *ring* must match *InputType2*, 3) the fourth domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.7 eWiseAdd() [5/8]

```
RC grb::eWiseAdd (
             Vector< OutputType, backend, Coords > & z,
             const Vector< MaskType, backend, Coords > & mask,
             const InputType1 alpha,
             const InputType2 beta,
             const Ring & ring = Ring(),
             const Phase & phase = EXECUTE,
             const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

Calculates the element-wise addition, $z+ = \alpha. + \beta$, under a given semiring, masked variant.

Note

This is an in-place operation.

Deprecated This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Note

A call to this function is equivalent to two in-place fold operations using the additive monoid of the given semiring. Please update any code that calls grb::eWiseAdd with such a sequence as soon as possible.

We may consider providing this function as an algorithm in the grb::algorithms namespace, similar to grb::algorithms::mpv. Please let the maintainers know if you would prefer such a solution over outright removal and replacement with two folds.

Template Parameters

descr	The descriptor to be used. Optional; default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise addition on.
InputType1	The left-hand side input type to the additive operator of the *ring*.
InputType2	The right-hand side input type to the additive operator of the *ring*.
OutputType	The result type of the additive operator of the *ring*.
MaskType	The nonzero type of the output mask vector.

Parameters

out	*z*	The output vector of type *Output↩ Type*. This may be a sparse vector.
in	*mask*	The output mask.
in	*alpha*	The left-hand input scalar of type *Input↩ Type1*.
in	*beta*	The right-hand input scalar of type *Input↩ Type2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH If *mask* and *z* do not have the same size.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the third domain of *ring* must match *InputType1*, 2) the fourth domain of *ring* must match *InputType2*, 3) the fourth domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.8 eWiseAdd() [6/8]

```
RC grb::eWiseAdd (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const InputType1 alpha,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

Calculates the element-wise addition, $z+ = \alpha . + y$, under a given semiring, masked variant.

Note

This is an in-place operation.

Deprecated This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Note

A call to this function is equivalent to two in-place fold operations using the additive monoid of the given semiring. Please update any code that calls grb::eWiseAdd with such a sequence as soon as possible.

We may consider providing this function as an algorithm in the grb::algorithms namespace, similar to grb::algorithms::mpv. Please let the maintainers know if you would prefer such a solution over outright removal and replacement with two folds.

Template Parameters

descr	The descriptor to be used. Optional; default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise addition on.
InputType1	The left-hand side input type to the additive operator of the *ring*.
InputType2	The right-hand side input type to the additive operator of the *ring*.
OutputType	The result type of the additive operator of the *ring*.
MaskType	The nonzero type of the output mask vector.

Parameters

out	*z*	The output vector of type *OutputType*. This may be a sparse vector.

Parameters

in	*mask*	The output mask.
in	*alpha*	The left-hand input scalar of type *Input↩ Type1*.
in	*y*	The right-hand input vector of type *Input↩ Type2*. This may be a sparse vector.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *y*, and *z* do not match. All input data containers are left untouched; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the third domain of *ring* must match *InputType1*, 2) the fourth domain of *ring* must match *InputType2*, 3) the fourth domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.9 eWiseAdd() [7/8]

```
RC grb::eWiseAdd (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const Vector< InputType1, backend, Coords > & x,
            const InputType2 beta,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

Calculates the element-wise addition, $z+ = x. + \beta$, under a given semiring, masked variant.

Note

This is an in-place operation.

Deprecated This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Note

A call to this function is equivalent to two in-place fold operations using the additive monoid of the given semiring. Please update any code that calls grb::eWiseAdd with such a sequence as soon as possible.

We may consider providing this function as an algorithm in the grb::algorithms namespace, similar to grb::algorithms::mpv. Please let the maintainers know if you would prefer such a solution over outright removal and replacement with two folds.

Template Parameters

descr	The descriptor to be used. Optional; default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise addition on.
InputType1	The left-hand side input type to the additive operator of the *ring*.
InputType2	The right-hand side input type to the additive operator of the *ring*.
OutputType	The result type of the additive operator of the *ring*.
MaskType	The nonzero type of the output mask vector.

Parameters

out	*z*	The output vector of type *OutputType*. This may be a sparse vector.
in	*mask*	The output mask.
in	*x*	The left-hand input vector of type *InputType1*. This may be a sparse vector.

Parameters

in	*beta*	The right-hand input scalar of type *Input↩ Type2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *x*, and *z* do not match. All input data containers are left untouched; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the third domain of *ring* must match *InputType1*, 2) the fourth domain of *ring* must match *InputType2*, 3) the fourth domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.10 eWiseAdd() [8/8]

```
RC grb::eWiseAdd (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

Calculates the element-wise addition of two vectors, $z+ = x. + y$, under a given semiring, masked variant.

Note

This is an in-place operation.

Deprecated This function has been deprecated since v0.5. It may be removed at latest at v1.0 of ALP/GraphBLAS– or any time earlier.

Note

A call to this function is equivalent to two in-place fold operations using the additive monoid of the given semiring. Please update any code that calls grb::eWiseAdd with such a sequence as soon as possible.

We may consider providing this function as an algorithm in the grb::algorithms namespace, similar to grb::algorithms::mpv. Please let the maintainers know if you would prefer such a solution over outright removal and replacement with two folds.

Template Parameters

descr	The descriptor to be used. Optional; default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise addition on.
InputType1	The left-hand side input type to the additive operator of the *ring*.
InputType2	The right-hand side input type to the additive operator of the *ring*.
OutputType	The result type of the additive operator of the *ring*.
MaskType	The nonzero type of the output mask vector.

Parameters

out	*z*	The output vector of type *OutputType*. This may be a sparse vector.
in	*mask*	The output mask vector of type *MaskType*.
in	*x*	The left-hand input vector of type *InputType1*. This may be a sparse vector.
in	*y*	The right-hand input vector of type *InputType2*. This may be a sparse vector.

Parameters

in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Note

There are also variants where *x* and/or *y* are scalars, as well as unmasked variants.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *x*, *y*, and *z* do not match. All input data containers are left untouched; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the third domain of *ring* must match *InputType1*, 2) the fourth domain of *ring* must match *InputType2*, 3) the fourth domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.11 eWiseApply() [1/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const InputType1 alpha,
            const InputType2 beta,
            const Monoid & monoid = Monoid(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >↩
::value, void >::type * const  = nullptr )
```

Computes $z = \alpha \odot \beta$, out of place, monoid version.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
Monoid	The monoid to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.

Parameters

out	*z*	The output vector.
in	*alpha*	The left-hand input scalar.

Parameters

in	*beta*	The right-hand input scalar.
in	*monoid*	The monoid with underlying operator ⊙.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Specialisation scalar inputs, unmasked monoid version.

A call to this function is equivalent to the following code:

```
typename OP::D3 tmp;
grb::apply( tmp, x, y, monoid.getOperator() );
grb::set( z, tmp, phase );
```

Returns

grb::SUCCESS On successful completion of this call.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.12 eWiseApply() [2/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const InputType1 alpha,
            const InputType2 beta,
            const OP & op = OP(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value,
void >::type * const  = nullptr )
```

Computes $z = \alpha \odot \beta$, out of place, operator version.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
OP	The operator to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.

Parameters

out	*z*	The output vector.
in	*alpha*	The left-hand input scalar.
in	*beta*	The right-hand input scalar.
in	*op*	The operator $\odot$.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Specialisation scalar inputs, unmasked operator version.

A call to this function is equivalent to the following code:

```
typename OP::D3 tmp;
grb::apply( tmp, x, y, op );
grb::set( z, tmp, phase );
```

Returns

grb::SUCCESS On successful completion of this call.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.13 eWiseApply() [3/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const InputType1 alpha,
            const Vector< InputType2, backend, Coords > & y,
            const Monoid & monoid = Monoid(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >↩
::value, void >::type * const  = nullptr )
```

Computes $z = \alpha \odot y$, out of place, monoid version.

Calculates the element-wise operation on one scalar to elements of one vector, $z = \alpha \odot y$, using the given operator. The input and output vectors must be of equal length.

For all indices *i* of *z*, its element z_i after the call to this function completes equals $\alpha \odot y_i$. Any old entries of *z* are removed.

After a successful call to this primitive, *z* shall be dense.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
Monoid	The monoid to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.

Parameters

out	*z*	The output vector.
in	*alpha*	The left-hand input scalar.
in	*y*	The right-hand input vector.
in	*monoid*	The monoid that provides the operator $\odot$.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *y* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.14 eWiseApply() [4/16]

```
RC grb::eWiseApply (
             Vector< OutputType, backend, Coords > & z,
             const InputType1 alpha,
             const Vector< InputType2, backend, Coords > & y,
             const OP & op = OP(),
             const Phase & phase = EXECUTE,
             const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value,
void >::type * const  = nullptr )
```

Computes $z = \alpha \odot y$, out of place, operator version.

Calculates the element-wise operation on one scalar to elements of one vector, $z = \alpha \odot y$, using the given operator. The input and output vectors must be of equal length.

For all indices *i* of *z*, its element z_i after the call to this function completes equals $\alpha \odot y_i$. Any old entries of *z* are removed. Entries *i* for which *y* has no nonzero will be skipped.

After a successful call to this primitive, the sparsity structure of *z* shall match that of *y*.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
OP	The operator to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.

Parameters

`out`	*z*	The output vector.
`in`	*alpha*	The left-hand input scalar.
`in`	*y*	The right-hand input vector.
`in`	*op*	The operator $\odot$.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *y* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.15 eWiseApply() [5/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< InputType1, backend, Coords > & x,
            const InputType2 beta,
            const Monoid & monoid = Monoid(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >↩
::value, void >::type * const  = nullptr )
```

Computes $z = x \odot \beta$, out of place, monoid variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = x \odot \beta$, using the given operator. The input and output vectors must be of equal length.

For all indices *i* of *z*, its element z_i after the call to this function completes equals $x_i \odot \beta$. Any old entries of *z* are removed.

After a successful call to this primitive, *z* shall be dense.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
Monoid	The monoid to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.

Parameters

out	*z*	The output vector.
in	*x*	The left-hand input vector.
in	*beta*	The right-hand input scalar.
in	*monoid*	The monoid that provides the operator $\odot$.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.16 eWiseApply() [6/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< InputType1, backend, Coords > & x,
            const InputType2 beta,
            const OP & op = OP(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value,
void >::type * const  = nullptr )
```

Computes $z = x \odot \beta$, out of place, operator variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = x.*\beta$, using the given operator. The input and output vectors must be of equal length.

For all valid indices *i* of *z*, its element z_i after the call to this function completes equals $x_i \odot \beta$. Any old entries of *z* are removed.

Entries *i* for which no nonzero exists in *x* are skipped. Therefore, after a successful call to this primitive, the nonzero structure of *z* will match that of *x*.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
OP	The operator to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.

Parameters

`out`	*z*	The output vector.
`in`	*x*	The left-hand input vector.
`in`	*beta*	The right-hand input scalar.
`in`	*op*	The operator $\odot$.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.17 eWiseApply() [7/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const Monoid & monoid = Monoid(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >↩
::value, void >::type * const  = nullptr )
```

Computes $z = x \odot y$, out of place, monoid variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = x \odot y$, using the given operator. The input and output vectors must be of equal length.

For all valid indices *i* of *z*, its element z_i after the call to this function completes equals $x_i \odot y_i$. Any old entries of *z* are removed.

After a successful call to this primitive, the nonzero structure of *z* will match that of the union of *x* and *y*. An implementing backend may skip processing indices *i* that are not in the union of the nonzero structure of *x* and *y*.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Monoid	The monoid to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.

Parameters

`out`	*z*	The output vector.
`in`	*x*	The left-hand input vector.
`in`	*y*	The right-hand input vector.
`in`	*monoid*	The monoid structure that $\odot$ corresponds to.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x*, *y* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.18 eWiseApply() [8/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const OP & op = OP(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value,
void >::type * const  = nullptr )
```

Computes $z = x \odot y$, out of place, operator variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = x \odot y$, using the given operator. The input and output vectors must be of equal length.

For all valid indices *i* of *z*, its element z_i after the call to this function completes equals $x_i \odot y_i$. Any old entries of *z* are removed. Entries *i* which have no nonzero in either *x* or *y* are skipped.

After a successful call to this primitive, the nonzero structure of *z* will match that of the intersection of *x* and *y*.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
OP	The operator to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.

Parameters

`out`	*z*	The output vector.
`in`	*x*	The left-hand input vector.
`in`	*y*	The right-hand input vector.
`in`	*op*	The operator $\odot$.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x*, *y* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.19 eWiseApply() [9/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const InputType1 alpha,
            const InputType2 beta,
            const Monoid & monoid = Monoid(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&grb::is_monoid< Monoid >::value, void >::type * const  = nullptr )
```

Computes $z = \alpha \odot \beta$, out of place, masked monoid version.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
Monoid	The monoid to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.
MaskType	The value type of the output mask vector.

Parameters

out	*z*	The output vector.
in	*mask*	The output mask.
in	*alpha*	The left-hand input scalar.
in	*beta*	The right-hand input scalar.
in	*monoid*	The monoid with underlying operator $\odot$.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Specialisation for scalar inputs, masked monoid version.

A call to this function is equivalent to the following code:

```
typename OP::D3 tmp;
grb::apply( tmp, alpha, beta, monoid.getOperator() );
grb::set( z, mask, tmp, phase );
```

Returns

grb::SUCCESS On successful completion of this call.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.20 eWiseApply() [10/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const InputType1 alpha,
            const InputType2 beta,
            const OP & op = OP(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&grb::is_operator< OP >::value, void >::type * const  = nullptr )
```

Computes $z = \alpha \odot \beta$, out of place, operator and masked version.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
OP	The operator to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.
MaskType	The value type of the output mask vector.

Parameters

out	*z*	The output vector.

Parameters

in	*mask*	The ouptut mask.
in	*alpha*	The left-hand input scalar.
in	*beta*	The right-hand input scalar.
in	*op*	The operator $\odot$.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Specialisation scalar inputs, masked operator version.

A call to this function is equivalent to the following code:

```
typename OP::D3 tmp;
grb::apply( tmp, x, y, op );
grb::set( z, mask, tmp, phase );
```

Returns

grb::SUCCESS On successful completion of this call.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.21 eWiseApply() [11/16]

```
RC grb::eWiseApply (
              Vector< OutputType, backend, Coords > & z,
              const Vector< MaskType, backend, Coords > & mask,
              const InputType1 alpha,
              const Vector< InputType2, backend, Coords > & y,
              const Monoid & monoid = Monoid(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&grb::is_monoid< Monoid >::value, void >::type * const  = nullptr )
```

Computes $z = \alpha \odot y$, out of place, masked monoid variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = \alpha \odot y$, using the given operator. The input and output vectors must be of equal length.

For all indices *i* of *z*, its element z_i after the call to this function completes equals $\alpha \odot y_i$. Any old entries of *z* are removed. Entries *i* for which *mask* evaluates `false` will be skipped.

After a successful call to this primitive, the sparsity structure of *z* shall match that of *mask* (after interpretation).

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
Monoid	The monoid to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.
MaskType	The value type of the output mask vector.

Parameters

out	*z*	The output vector.
out	*mask*	The output mask.
in	*alpha*	The left-hand input scalar.
in	*y*	The right-hand input vector.
in	*monoid*	The monoid that provides the operator $\odot$.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, a y and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.22 eWiseApply() [12/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const InputType1 alpha,
            const Vector< InputType2, backend, Coords > & y,
            const OP & op = OP(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&grb::is_operator< OP >::value, void >::type * const  = nullptr )
```

Computes $z = \alpha \odot y$, out of place, masked operator version.

Calculates the element-wise operation on one scalar to elements of one vector, $z = \alpha \odot y$, using the given operator. The input and output vectors must be of equal length.

For all indices *i* of *z*, its element z_i after the call to this function completes equals $\alpha \odot y_i$. Any old entries of *z* are removed. Entries *i* for which *y* has no nonzero will be skipped, as will entries *i* for which *mask* evaluates `false`.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
OP	The operator to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.
MaskType	The value type of the mask vector.

Parameters

out	*z*	The output vector.
in	*mask*	The output mask.
in	*alpha*	The left-hand input scalar.
in	*y*	The right-hand input vector.
in	*op*	The operator $\odot$.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *y* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.23 eWiseApply() [13/16]

```
RC grb::eWiseApply (
              Vector< OutputType, backend, Coords > & z,
              const Vector< MaskType, backend, Coords > & mask,
              const Vector< InputType1, backend, Coords > & x,
              const InputType2 beta,
              const Monoid & monoid = Monoid(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&grb::is_monoid< Monoid >::value, void >::type * const  = nullptr )
```

Computes $z = x \odot \beta$, out of place, masked monoid variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = x \odot \beta$, using the given operator. The input and output vectors must be of equal length.

For all indices *i* of *z*, its element z_i after the call to this function completes equals $x_i \odot \beta$. Any old entries of *z* are removed. Entries *i* for which *mask* evaluates `false` will be skipped.

After a successful call to this primitive, the sparsity structure of *z* matches that of *mask* (after interpretation).

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
Monoid	The monoid to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.
MaskType	The value type of the mask vector.

Parameters

out	*z*	The output vector.
out	*mask*	The output mask.
in	*x*	The left-hand input vector.
in	*beta*	The right-hand input scalar.
in	*monoid*	The monoid that provides the operator $\odot$.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *x* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.24 eWiseApply() [14/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const Vector< InputType1, backend, Coords > & x,
            const InputType2 beta,
            const OP & op = OP(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&grb::is_operator< OP >::value, void >::type * const  = nullptr )
```

Computes $z = x \odot \beta$, out of place, masked operator variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = x.*\beta$, using the given operator. The input and output vectors must be of equal length.

For all valid indices *i* of *z*, its element z_i after the call to this function completes equals $x_i \odot \beta$. Any old entries of *z* are removed.

Entries *i* for which no nonzero exists in *x* are skipped. Entries *i* for which the mask evaluates `false` are skipped as well.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Equal to descriptors::no_operation if left unspecified.
OP	The operator to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the output vector.
MaskType	The value type of the output mask vector.

Parameters

out	*z*	The output vector.
in	*mask*	The output mask.
in	*x*	The left-hand input vector.
in	*beta*	The right-hand input scalar.
in	*op*	The operator $\odot$.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *x*, and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.25 eWiseApply() [15/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const Monoid & monoid = Monoid(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&grb::is_monoid< Monoid >::value, void >::type * const  = nullptr )
```

Computes $z = x \odot y$, out of place, masked monoid variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = x \odot y$, using the given operator. The input and output vectors must be of equal length.

For all valid indices *i* of *z*, its element z_i after the call to this function completes equals $x_i \odot y_i$. Any old entries of *z* are removed. Entries *i* for which *mask* evaluates `false` will be skipped.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Monoid	The monoid to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.
MaskType	The value type of the mask vector.

Parameters

`out`	*z*	The output vector.

Parameters

in	*mask*	The output mask.
in	*x*	The left-hand input vector.
in	*y*	The right-hand input vector.
in	*monoid*	The monoid structure that $\odot$ corresponds to.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x*, *y* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.26 eWiseApply() [16/16]

```
RC grb::eWiseApply (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const OP & op = OP(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&grb::is_operator< OP >::value, void >::type * const  = nullptr )
```

Computes $z = x \odot y$, out of place, masked operator variant.

Calculates the element-wise operation on one scalar to elements of one vector, $z = x \odot y$, using the given operator. The input and output vectors must be of equal length.

For all valid indices *i* of *z*, its element z_i after the call to this function completes equals $x_i \odot y_i$. Any old entries of *z* are removed. Entries *i* which have no nonzero in either *x* or *y* are skipped, as will entries *i* for which *mask* evaluates `false`.

After a successful call to this primitive, the nonzero structure of *z* will match that of the intersection of *x* and *y*.

Note

When applying element-wise operators on sparse vectors using semirings, there is a difference between interpreting missing values as an annihilating identity or as a neutral identity– intuitively, such identities are known as 'zero' or 'one', respectively. As a consequence, there are two different variants for element-wise operations whose names correspond to their intuitive meanings:

- grb::eWiseAdd (neutral), and
- grb::eWiseMul (annihilating). The above two primitives require a semiring. The same functionality is provided by grb::eWiseApply depending on whether a monoid or operator is provided:
- grb::eWiseApply using monoids (neutral),
- grb::eWiseApply using operators (annihilating).

However, grb::eWiseAdd and grb::eWiseMul provide in-place semantics, while grb::eWiseApply does not.

An grb::eWiseAdd with some semiring and a grb::eWiseApply using its additive monoid thus are equivalent if operating when operating on empty outputs.

An grb::eWiseMul with some semiring and a grb::eWiseApply using its multiplicative operator thus are equivalent when operating on empty outputs.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
OP	The operator to use.
InputType1	The value type of the left-hand vector.
InputType2	The value type of the right-hand scalar.
OutputType	The value type of the ouput vector.
MaskType	The value type of the output mask vector.

Parameters

out	*z*	The output vector.
in	*mask*	The output mask.
in	*x*	The left-hand input vector.
in	*y*	The right-hand input vector.
in	*op*	The operator $\odot$.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *x*, *y*, and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.27 eWiseLambda()

```
RC grb::eWiseLambda (
            const Func f,
            const Vector< DataType, backend, Coords > & x,
            Args...    )
```

Executes an arbitrary element-wise user-defined function *f* on any number of vectors of equal length.

Warning

This is a relatively advanced function. It is recommended to read this specifications and its warnings before using it, or to instead exclusively only use the other primitives in Level-1 Primitives.

The vectors touched by *f* can be accessed in a read-only or a read/write fashion. The function *f* must be parametrised in a global index *i*, and *f* is only allowed to access elements of the captured vectors *on that specific index*.

Warning

Any attempt to access a vector element at a position differing from *i* will result in undefined behaviour.

All vectors captured by *f* must furthermore all be given as additional (variadic) arguments to this primitive. Captured vectors can only be used for dereferencing elements at a given position *i*; any other use invokes undefined behaviour.

Warning

In particular, captured vectors may not be passed to other ALP/GraphBLAS primitives *within f*.

This primitive will execute *f* on all indices where the first given such vector argument has nonzeroes. All other indices *i* will be ignored.

Warning

Therefore, for containers of which *f* references the *i-th* element, must indeed have a nonzero at position *i* or otherwise undefined behaviour is invoked.

This primitive hence allows a user to implement any level-1 like BLAS functionality over any number of input/output vectors, and also allows to compute multiple level-1 (like) BLAS functionalities as a single pass over the involved containers.

Note

Since the introduction of the nonblocking backend, rewriting *f* in terms of native ALP/GraphBLAS primitives no longer implies performance penalties (when compiling for the nonblocking backend)– rather, the nonblocking backend is likely to do better than manually fusing multiple level-1 like operations using this primitive, especially when the captured vectors are small relative to the private caches on the target architecture.

The function *f* may also capture scalars for read-only access.

Note

As a convention, consider always passing scalars by value, since otherwise the compilation of your code with a non-blocking backend may (likely) result in data races.

If grb::Properties::writableCaptured evaluates `true` then captured scalars may also safely be written to, instead of requiring to be read-only.

Note

This is useful for fusing reductions within other level-1 like operations.

Warning

If updating scalars using this primitive, be aware that the updates are local to the current user process only.

Note

If, after execution of this primitive, an updated scalar is expected to be synchronised across all user processes, see grb::collectives.

As a rule of thumb, parallel GraphBLAS implementations, due to being data-centric, *cannot* support writeable scalar captures and will have grb::Properties::writableCaptured evaluate to `false`.

A portable ALP/GraphBLAS algorithm should therefore either not rely on read/write captured scalars passed to this primitive, *or* provide different code paths to handle the two cases of the grb::Properties::writableCaptured backend property.

If the above sounds too tedious, consider rewriting *f* in terms of native ALP/GraphBLAS functions, with the scalar reductions performed by the scalar variants of grb::foldl and grb::foldr, e.g.

Warning

When compiling with a blocking backend, rewriting *f* in terms of native GraphBLAS primitives typically results in a slowdown due to this primitive naturally fusing potentially multiple operations together (which was the original motivation of Yzelman et al., 2020 for introducing this primitive. Rewriting *f* into a (sequence of) native GraphBLAS primtives does *not* carry a performance when compiling with a nonblocking backend, however.

Note

This is an addition to the GraphBLAS C specification. It is alike user-defined operators, monoids, and semirings, except that this primitive allows execution on arbitrarily many inputs and arbitrarily many outputs.

Template Parameters

Func	the user-defined lambda function type.
DataType	the type of the user-supplied vector example.
backend	the backend type of the user-supplied vector example.

Parameters

Parameters

in	*f*	The user-supplied lambda. This lambda should only capture and reference vectors of the same length as *x*. The lambda function should prescribe the operations required to execute at a given index *i*. Captured ALP/↩ Graph↩ BLAS vectors can access that element via the operator[]. It is illegal to access any element not at position *i*. The lambda takes only

Parameters

in	*x*	The vector the lambda will be executed on. This argument determines which indices *i* will be accessed during the elementwise operation–elements with indices *i* that do not appear in *x* will be skipped during evaluation of *f*.

The remaining arguments must collect all vectors the lambda is to access elements of. Such vectors must be of the same length as *x*. If this constraint is violated, grb::MISMATCH shall be returned.

Note

These are passed using variadic arguments and so can contain any number of containers of type grb::Vector.

Distributed-memory ALP/GraphBLAS backends, apart from performing dimension checking, may also require data redistribution in case that different vectors are distributed differently.

Warning

Using a grb::Vector inside a lambda passed to this function while not passing that same vector into its variadic argument list, will result in undefined behaviour.

Due to the constraints on *f* described above, it is illegal to capture some vector *y* and have the following line in the body of *f:* `x[i] += x[i+1]`. Vectors can only be dereferenced at position *i* and *i* alone.

Returns

grb::SUCCESS When the lambda is successfully executed.

grb::MISMATCH When two or more vectors passed to *args* are not of equal length.

grb::PANIC When ALP/GraphBLAS has encountered an unrecoverable error. The state of ALP becomes undefined after having returned this error code, and users can only attempt to exit the application gracefully.

Example.

An example valid use:

```
void f(
     double &alpha,
     grb::Vector< double > &y,
     const double beta,
     const grb::Vector< double > &x,
     const grb::Semiring< double > ring
) {
     assert( grb::size(x) == grb::size(y) );
     assert( grb::nnz(x) == grb::size(x) );
     assert( grb::nnz(y) == grb::size(y) );
     alpha = ring.getZero();
     grb::eWiseLambda(
         [&alpha,beta,&x,&y,ring]( const size_t i ) {
             double mul;
             const auto mul_op = ring.getMultiplicativeOperator();
             const auto add_op = ring.getAdditiveOperator();
             grb::apply( y[i], beta, x[i], mul_op );
             grb::apply( mul, x[i], y[i], mul_op );
             grb::foldl( alpha, mul, add_op );
     }, x, y );
     grb::collectives::allreduce( alpha, add_op );
}
```

This code takes a value *beta*, a vector *x*, and a semiring *ring* and computes: 1) *y* as the element-wise multiplication (under *ring*) of *beta* and *x*; and 2) *alpha* as the dot product (under *ring*) of *x* and *y*. This function can easily be made agnostic to whatever exact semiring is used by templating the type of *ring*. As it is, this code is functionally equivalent to:

```
grb::eWiseMul( y, beta, x, ring );
grb::dot( alpha, x, y, ring );
```

If the latter code block is compiled using a blocking ALP/GraphBLAS backend, the version using the lambdas is expected to execute faster as both *x* and *y* are streamed only once, while the latter code may stream both vectors twice. This performance difference disappears when compiling the latter code block using a nonblocking backend instead.

Warning

The following code is invalid:

```
template< class Operator >
void f(
     grb::Vector< double > &x,
     const Operator op
) {
     grb::eWiseLambda(
         [&x,&op]( const size_t i ) {
             grb::apply( x[i], x[i], x[i+1], op );
     }, x );
}
```

Only a Vector::lambda_reference to position exactly equal to *i* may be used within this function.

See also

Vector::operator[ ]()

Vector::lambda_reference

Performance semantics

Each backend must define performance semantics for this primitive. It is expected that the defined performance semantics depend on the given lambda function *f*, the size of the containers passed into this primitive, as well as how many containers are passed into this primitive.

See also

Performance Semantics

7.11.3.28 eWiseMul() [1/8]

```
RC grb::eWiseMul (
            Vector< OutputType, backend, Coords > & z,
            const InputType1 alpha,
            const InputType2 beta,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

In-place element-wise multiplication of two scalars, $z+ = \alpha. * \beta$, under a given semiring.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise multiply with.
InputType1	The left-hand side input type.
InputType2	The right-hand side input type.
OutputType	The output type.

Parameters

out	*z*	The output vector of type *Output↩ Type*.
in	*alpha*	The left-hand input scalar of type *Input↩ Type1*.

Parameters

in	*beta*	The right-hand input scalar of type *Input↩ Type2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Warning

Unlike grb::eWiseApply using monoids, given sparse vectors, missing elements in sparse input vectors are now interpreted as a the zero identity, therefore annihilating instead of acting as a monoid identity. Therefore even when *z* is empty on input, the grb::eWiseApply with monoids does not incur the same behaviour as this function. The grb::eWiseApply with operators *is* similar, except that this function is in-place and grb::eWiseApply is not.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting, and
- grb::descriptors::dense.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *ring* must match *InputType1*, 2) the second domain of *ring* must match *InputType2*, 3) the third domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.29 eWiseMul() [2/8]

```
RC grb::eWiseMul (
            Vector< OutputType, backend, Coords > & z,
            const InputType1 alpha,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

In-place element-wise multiplication of a scalar and vector, $z+ = \alpha . * y$, under a given semiring.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise multiply with.
InputType1	The left-hand side input type.
InputType2	The right-hand side input type.
OutputType	The output type.

Parameters

out	*z*	The output vector of type *Output↩ Type*.
in	*alpha*	The left-hand input scalar of type *Input↩ Type1*.
in	*y*	The right-hand input vector of type *Input↩ Type2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *y* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Warning

Unlike grb::eWiseApply using monoids, given sparse vectors, missing elements in sparse input vectors are now interpreted as a the zero identity, therefore annihilating instead of acting as a monoid identity. Therefore even when *z* is empty on input, the grb::eWiseApply with monoids does not incur the same behaviour as this function. The grb::eWiseApply with operators *is* similar, except that this function is in-place and grb::eWiseApply is not.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting, and
- grb::descriptors::dense.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *ring* must match *InputType1*, 2) the second domain of *ring* must match *InputType2*, 3) the third domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.30 eWiseMul() [3/8]

```
RC grb::eWiseMul (
            Vector< OutputType, backend, Coords > & z,
            const Vector< InputType1, backend, Coords > & x,
            const InputType2 beta,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

In-place element-wise multiplication of a vector and scalar, $z+ = x. * \beta$, under a given semiring.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise multiply with.
InputType1	The left-hand side input type.
InputType2	The right-hand side input type.
OutputType	The output type.

Parameters

out	*z*	The output vector of type *OutputType*.
in	*x*	The left-hand input vector of type *InputType1*.
in	*beta*	The right-hand input scalar of type *InputType2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Warning

Unlike grb::eWiseApply using monoids, given sparse vectors, missing elements in sparse input vectors are now interpreted as a the zero identity, therefore annihilating instead of acting as a monoid identity. Therefore even when *z* is empty on input, the grb::eWiseApply with monoids does not incur the same behaviour as this function. The grb::eWiseApply with operators *is* similar, except that this function is in-place and grb::eWiseApply is not.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting, and
- grb::descriptors::dense.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *ring* must match *InputType1*, 2) the second domain of *ring* must match *InputType2*, 3) the third domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.31 eWiseMul() [4/8]

```
RC grb::eWiseMul (
            Vector< OutputType, backend, Coords > & z,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

In-place element-wise multiplication of two vectors, $z+ = x. * y$, under a given semiring.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise multiply with.
InputType1	The left-hand side input type.
InputType2	The right-hand side input type.
OutputType	The output type.

Parameters

out	*z*	The output vector of type *Output↩ Type*.
in	*x*	The left-hand input vector of type *Input↩ Type1*.
in	*y*	The right-hand input vector of type *Input↩ Type2*.

Parameters

in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *x*, *y*, and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Warning

Unlike grb::eWiseApply using monoids, given sparse vectors, missing elements in sparse input vectors are now interpreted as a the zero identity, therefore annihilating instead of acting as a monoid identity. Therefore even when *z* is empty on input, the grb::eWiseApply with monoids does not incur the same behaviour as this function. The grb::eWiseApply with operators *is* similar, except that this function is in-place and grb::eWiseApply is not.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting, and
- grb::descriptors::dense.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *ring* must match *InputType1*, 2) the second domain of *ring* must match *InputType2*, 3) the third domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.32 eWiseMul() [5/8]

```
RC grb::eWiseMul (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const InputType1 alpha,
            const InputType2 beta,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

In-place element-wise multiplication of two scalars, $z+ = \alpha. * \beta$, under a given semiring, masked variant.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise multiply with.
InputType1	The left-hand side input type.
InputType2	The right-hand side input type.
OutputType	The output vector type.
MaskType	The output mask type.

Parameters

in,out	*z*	The output vector of type *OutputType*.
in	*mask*	The ouput mask of type *MaskType*.
in	*alpha*	The left-hand input scalar of type *InputType1*.
in	*beta*	The right-hand input scalar of type *InputType2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH If *mask* and *z* have different size.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Warning

Unlike grb::eWiseApply using monoids, given sparse vectors, missing elements in sparse input vectors are now interpreted as a the zero identity, therefore annihilating instead of acting as a monoid identity. Therefore even when *z* is empty on input, the grb::eWiseApply with monoids does not incur the same behaviour as this function. The grb::eWiseApply with operators *is* similar, except that this function is in-place and grb::eWiseApply is not.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *ring* must match *InputType1*, 2) the second domain of *ring* must match *InputType2*, 3) the third domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.33 eWiseMul() [6/8]

```
RC grb::eWiseMul (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const InputType1 alpha,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

In-place element-wise multiplication of a scalar and vector, $z+ = \alpha . * y$, under a given semiring, masked variant.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise multiply with.
InputType1	The left-hand side input type.
InputType2	The right-hand side input type.
OutputType	The output vector type.
MaskType	The output mask type.

Parameters

in,out	*z*	The output vector of type *OutputType*.
in	*mask*	The ouput mask of type *MaskType*.
in	*alpha*	The left-hand input scalar of type *InputType1*.
in	*y*	The right-hand input vector of type *InputType2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *y*, and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Warning

Unlike grb::eWiseApply using monoids, given sparse vectors, missing elements in sparse input vectors are now interpreted as a the zero identity, therefore annihilating instead of acting as a monoid identity. Therefore even when *z* is empty on input, the grb::eWiseApply with monoids does not incur the same behaviour as this function. The grb::eWiseApply with operators *is* similar, except that this function is in-place and grb::eWiseApply is not.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *ring* must match *InputType1*, 2) the second domain of *ring* must match *InputType2*, 3) the third domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.34 eWiseMul() [7/8]

```
RC grb::eWiseMul (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const Vector< InputType1, backend, Coords > & x,
            const InputType2 beta,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

In-place element-wise multiplication of a vector and scalar, $z+ = x. * \beta$, under a given semiring, masked variant.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise multiply with.
InputType1	The left-hand side input type.
InputType2	The right-hand side input type.
OutputType	The output vector type.
MaskType	The output mask type.

Parameters

in,out	*z*	The output vector of type *Output*↩ *Type*.
in	*mask*	The output mask of type *Mask*↩ *Type*.

Parameters

in	*x*	The left-hand input vector of type *Input↩ Type1*.
in	*beta*	The right-hand input scalar of type *Input↩ Type2*.
in	*ring*	The generalized semiring under which to perform this element-wise multiplication.
in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *x* and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Warning

Unlike grb::eWiseApply using monoids, given sparse vectors, missing elements in sparse input vectors are now interpreted as a the zero identity, therefore annihilating instead of acting as a monoid identity. Therefore even when *z* is empty on input, the grb::eWiseApply with monoids does not incur the same behaviour as this function. The grb::eWiseApply with operators *is* similar, except that this function is in-place and grb::eWiseApply is not.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *ring* must match *InputType1*, 2) the second domain of *ring* must match *InputType2*, 3) the third domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.35 eWiseMul() [8/8]

```
RC grb::eWiseMul (
            Vector< OutputType, backend, Coords > & z,
            const Vector< MaskType, backend, Coords > & mask,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const Ring & ring = Ring(),
            const Phase & phase = EXECUTE,
            const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >↩
::value, void >::type * const  = nullptr )
```

In-place element-wise multiplication of two vectors, $z+ = x. * y$, under a given semiring, masked variant.

Template Parameters

descr	The descriptor to be used. Optional; the default is grb::descriptors::no_operation.
Ring	The semiring type to perform the element-wise multiply with.
InputType1	The left-hand side input type.
InputType2	The right-hand side input type.
OutputType	The output vector type.
MaskType	The output mask type.

Parameters

`in,out`	*z*	The output vector of type *OutputType*.
`in`	*mask*	The ouput mask of type *MaskType*.
`in`	*x*	The left-hand input vector of type *InputType1*.
`in`	*y*	The right-hand input vector of type *InputType2*.
`in`	*ring*	The generalized semiring under which to perform this element-wise multiplication.

Parameters

in	*phase*	The grb::Phase the call should execute. Optional; the default parameter is grb::EXECUTE.

Returns

grb::SUCCESS On successful completion of this call.

grb::MISMATCH Whenever the dimensions of *mask*, *x*, *y*, and *z* do not match. All input data containers are left untouched if this exit code is returned; it will be as though this call was never made.

grb::FAILED If *phase* is grb::EXECUTE, indicates that the capacity of *z* was insufficient. The output vector *z* is cleared, and the call to this function has no further effects.

grb::OUTOFMEM If *phase* is grb::RESIZE, indicates an out-of-memory exception. The call to this function shall have no other effects beyond returning this error code; the previous state of *z* is retained.

grb::PANIC A general unmitigable error has been encountered. If returned, ALP enters an undefined state and the user program is encouraged to exit as quickly as possible.

Warning

Unlike grb::eWiseApply using monoids, given sparse vectors, missing elements in sparse input vectors are now interpreted as a the zero identity, therefore annihilating instead of acting as a monoid identity. Therefore even when *z* is empty on input, the grb::eWiseApply with monoids does not incur the same behaviour as this function. The grb::eWiseApply with operators *is* similar, except that this function is in-place and grb::eWiseApply is not.

Valid descriptors

- grb::descriptors::no_operation,
- grb::descriptors::no_casting,
- grb::descriptors::dense,
- grb::descriptors::invert_mask,
- grb::descriptors::structural, and
- grb::descriptors::structural_complement.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *ring* must match *InputType1*, 2) the second domain of *ring* must match *InputType2*, 3) the third domain of *ring* must match *OutputType*. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.36 foldl() [1/3]

```
RC grb::foldl (
            IOType & x,
            const Vector< InputType, backend, Coords > & y,
            const Monoid & monoid = Monoid(),
            const typename std::enable_if< !grb::is_object< IOType >::value &&grb::is_monoid<
Monoid >::value, void >::type * const  = nullptr )
```

Folds a vector into a scalar, left-to-right.

Unmasked monoid variant. See masked variant for the full documentation.

7.11.3.37 foldl() [2/3]

```
RC grb::foldl (
            IOType & x,
            const Vector< InputType, backend, Coords > & y,
            const Vector< MaskType, backend, Coords > & mask,
            const Monoid & monoid = Monoid(),
            const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object<
InputType >::value &&!grb::is_object< MaskType >::value &&grb::is_monoid< Monoid >::value,
void >::type * const  = nullptr )
```

Reduces, or *folds*, a vector into a scalar.

Reduction takes place according a monoid $(\oplus, 1)$, where $\oplus : D_1 \times D_2 \to D_3$ with associated identities $1_k in D_k$. Usually, $D_k \subseteq D_3, 1 \leq k < 3$, though other more exotic structures may be envisioned (and used).

Let $x_0 = 1$ and let $x_{i+1} = \begin{cases} x_i \oplus y_i \text{ if } y_i \text{ is nonzero and } m_i \text{ evaluates true} \\ x_i \text{ otherwise} \end{cases}$, for all $i \in \{0, 1, \ldots, n-1\}$.

Note

Per this definition, the folding happens in a left-to-right direction. If another direction is wanted, which may have use in cases where D_1 differs from D_2, then either a monoid with those operator domains switched may be supplied, or grb::foldr may be used instead.

After a successfull call, *x* will be equal to x_n.

Note that the operator $\oplus$ must be associative since it is part of a monoid. This algebraic property is exploited when parallelising the requested operation. The identity is required when parallelising over multiple user processes.

Warning

In so doing, the order of the evaluation of the reduction operation should not be expected to be a serial, left-to-right, evaluation of the computation chain.

Template Parameters

descr	The descriptor to be used (descriptors::no_operation if left unspecified).
Monoid	The monoid to use for reduction.
InputType	The type of the elements in the supplied ALP/GraphBLAS vector *y*.
IOType	The type of the output scalar *x*.
MaskType	The type of the elements in the supplied ALP/GraphBLAS vector *mask*.

Parameters

out	*x*	The result of the reduction.
in	*y*	Any ALP/GraphBLAS vector. This vector may be sparse.
in	*mask*	Any ALP/GraphBLAS vector. This vector may be sparse.
in	*monoid*	The monoid under which to perform this reduction.

Returns

grb::SUCCESS When the call completed successfully.

grb::MISMATCH If a *mask* was not empty and does not have size equal to *y*.

grb::ILLEGAL If the provided input vector *y* was not dense, while grb::descriptors::dense was given.

See also

grb::foldr provides similar in-place functionality.

grb::eWiseApply provides out-of-place semantics.

Valid descriptors

grb::descriptors::no_operation, grb::descriptors::no_casting, grb::descriptors::dense, grb::descriptors::invert_mask, grb::descriptors::structural, grb::descriptors::structural_complement

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is given, then 1) the first domain of *monoid* must match *InputType*, 2) the second domain of *op* must match *IOType*, 3) the third domain must match *IOType*, and 4) the element type of *mask* must be `bool`. If one of these is not true, the code shall not compile.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.11.3.38 foldl() [3/3]

```
RC grb::foldl (
              IOType & x,
              const Vector< InputType, backend, Coords > & y,
              const Vector< MaskType, backend, Coords > & mask,
              const OP & op = OP(),
              const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object<
MaskType >::value &&grb::is_operator< OP >::value, void >::type * const  = nullptr )
```

Folds a vector into a scalar, left-to-right.

Unmasked operator variant. See masked variant for the full documentation.

Deprecated This signature is deprecated. It was implemented for reference (and reference_omp), but could not be implemented for BSP1D and other distributed-memory backends. This signature may be removed with any release beyond 0.6.

7.11.3.39 foldr() [1/2]

```
RC grb::foldr (
              const Vector< InputType, backend, Coords > & x,
              const Vector< MaskType, backend, Coords > & mask,
              IOType & y,
              const Monoid & monoid = Monoid(),
              const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object<
InputType >::value &&!grb::is_object< MaskType >::value &&grb::is_monoid< Monoid >::value,
void >::type * const  = nullptr )
```

Folds a vector into a scalar, right-to-left.

Masked variant. See the masked, left-to-right variant for the full documentation.

7.11.3.40 foldr() [2/2]

```
RC grb::foldr (
              const Vector< InputType, backend, Coords > & y,
              IOType & x,
              const Monoid & monoid = Monoid(),
              const typename std::enable_if< !grb::is_object< IOType >::value &&grb::is_monoid<
Monoid >::value, void >::type * const  = nullptr )
```

Folds a vector into a scalar, right-to-left.

Unmasked variant. See the masked, left-to-right variant for the full documentation.

7.12 Level-2 Primitives

A collection of functions that allow GraphBLAS operators, monoids, and semirings work on a mix of zero-dimensional, one-dimensional, and two-dimensional containers.

Functions

- template<typename Func , typename DataType , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_backend, typename... Args>
  RC eWiseLambda (const Func f, const Matrix< DataType, implementation, RIT, CIT, NIT > &A, Args...)

  Executes an arbitrary element-wise user-defined function f on all nonzero elements of a given matrix A.
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place doubly-masked sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Matrix< InputType2, backend, RIT, NIT, CIT > &A, const Vector< InputType1, backend, Coords > &v, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place masked sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.
- template<Descriptor descr = descriptors::no_operation, class Semiring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &u_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Semiring &semiring=Semiring(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_semiring< Semiring >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value, void >::type *const =nullptr)

  Right-handed in-place doubly-masked sparse matrix times vector multiplication, $u = u + Av$.
- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_backend>
  RC mxv (Vector< IOType, implementation, Coords > &u, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Vector< InputType1, implementation, Coords > &v, const Ring &ring, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Right-handed in-place sparse matrix–vector multiplication, $u = u + Av$, over a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , typename Coords , enum Backend implementation = config::default_backend>
  RC mxv (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords > &mask, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Vector< InputType1, implementation, Coords > &v, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Right-handed in-place masked sparse matrix–vector multiplication, $u = u + Av$, over a given semiring.
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType1, backend, Coords > &v, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

Left-handed in-place sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Left-handed in-place doubly-masked sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

- template<Descriptor descr = descriptors::no_operation, class Semiring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &u_mask, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Semiring &semiring=Semiring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Semiring >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Left-handed in-place doubly-masked sparse matrix times vector multiplication, $u = u + vA$.

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend implementation = config::default_backend>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Left-handed in-place sparse matrix–vector multiplication, $u = u + vA$, over a given semiring.

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , Backend implementation>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords > &mask, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *=nullptr)

  Left-handed in-place masked sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend implementation = config::default_backend>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords > &mask, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Left-handed in-place masked sparse matrix–vector multiplication, $u = u + vA$, over a given semiring.

7.12.1 Detailed Description

A collection of functions that allow GraphBLAS operators, monoids, and semirings work on a mix of zero-dimensional, one-dimensional, and two-dimensional containers.

That is, these functions allow various linear algebra operations on scalars, objects of type grb::Vector, and objects of type grb::Matrix.

Note

The backends of each opaque data type should match.

7.12.2 Function Documentation

7.12.2.1 eWiseLambda()

```
RC grb::eWiseLambda (
            const Func f,
            const Matrix< DataType, implementation, RIT, CIT, NIT > & A,
            Args...    )
```

Executes an arbitrary element-wise user-defined function *f* on all nonzero elements of a given matrix *A*.

The user-defined function is passed as a lambda which can capture whatever the user would like, including one or multiple grb::Vector instances, or multiple scalars. When capturing vectors, these should also be passed as a additional arguments to this functions so to make sure those vectors are synchronised for access on all row- and column- indices corresponding to locally stored nonzeroes of *A*.

Only the elements of a single matrix may be iterated upon.

Note

Rationale: while it is reasonable to expect an implementation be able to synchronise vector elements, it may be unreasonable to expect two different matrices can be jointly accessed via arbitrary lambda functions.

Warning

The lambda shall only be executed on the data local to the user process calling this function! This is different from the various fold functions, or grb::dot, in that the semantics of those functions always result in globally synchronised result. To achieve the same effect with user-defined lambdas, the users should manually prescribe how to combine the local results into global ones, for instance, by subsequent calls to grb::collectives.

Note

This is an addition to the GraphBLAS. It is alike user-defined operators, monoids, and semirings, except it allows execution on arbitrarily many inputs and arbitrarily many outputs.

Template Parameters

Func	the user-defined lambda function type.
DataType	the type of the user-supplied matrix.
backend	the backend type of the user-supplied vector example.

Parameters

Parameters

in	*f*	The user-supplied lambda. This lambda should only capture and reference vectors of the same length as either the row or column dimension length of *A*. The lambda function should prescribe the operations required to execute on a given reference to a matrix nonzero of *A* (of type *Data↩ Type*) at a given index (i, j).
		Captured Graph↩ BLAS

Parameters

in	*A*	The matrix the lambda is to access the elements of.

The remainder arguments should enumerate all vectors the lambda is to access elements of. Each such vector must be of the same length as *nrows(A)* or *ncols(A)*. If this constraint is violated, grb::MISMATCH shall be returned. If a given vector length equals *nrows(A)*, the vector shall be synchronized for access on *i*. If the vector length equals *ncols(A)*, the vector shall be synchronized for access on *j*. If *A* is square, the vectors will be synchronised for access on both *i and j*.

Note

These vectors are passed using a variadic argument list and so may contain any number of containers of type grb::Vector, potentially with differing nonzero types, as separate arguments.

Warning

Using a grb::Vector inside a lambda passed to this function while not passing that same vector into the variadic argument list will result in undefined behaviour.

Due to the constraints on *f* described above, it is illegal to capture some vector *y* and have the following line in the body of *f:* `x[i] += x[i+1]`. Vectors can only be dereferenced at position *i* and *i* alone, and similarly for access using *j*. For square matrices, however, the following code in the body is accepted, however: `x[i] += x[j]`.

Returns

grb::SUCCESS When the lambda is successfully executed.

grb::MISMATCH When two or more vectors passed into the variadic argument list are not of appropriate length.

Warning

Captured scalars will be local to the user process executing the lambda. To retrieve the global dot product, an allreduce must explicitly be called.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.2 mxv() [1/6]

```
RC grb::mxv (
              Vector< IOType, backend, Coords > & u,
              const Matrix< InputType2, backend, RIT, CIT, NIT > & A,
              const Vector< InputType1, backend, Coords > & v,
              const AdditiveMonoid & add = AdditiveMonoid(),
              const MultiplicativeOperator & mul = MultiplicativeOperator(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator<
MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< Input↩
Type1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >↩
::value, void >::type * const  = nullptr )
```

Right-handed in-place sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.

See the documentation of grb::vxm for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.3 mxv() [2/6]

```
RC grb::mxv (
              Vector< IOType, backend, Coords > & u,
              const Vector< InputType3, backend, Coords > & mask,
              const Matrix< InputType2, backend, RIT, CIT, NIT > & A,
              const Vector< InputType1, backend, Coords > & v,
              const Vector< InputType4, backend, Coords > & v_mask,
              const AdditiveMonoid & add = AdditiveMonoid(),
              const MultiplicativeOperator & mul = MultiplicativeOperator(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator<
MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< Input↩
Type1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value
&&!grb::is_object< InputType4 >::value &&!std::is_same< InputType2, void >::value, void >↩
::type * const  = nullptr )
```

Right-handed in-place doubly-masked sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.

See the documentation of grb::mxv for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.4 mxv() [3/6]

```
RC grb::mxv (
              Vector< IOType, backend, Coords > & u,
              const Vector< InputType3, backend, Coords > & mask,
              const Matrix< InputType2, backend, RIT, NIT, CIT > & A,
              const Vector< InputType1, backend, Coords > & v,
              const AdditiveMonoid & add = AdditiveMonoid(),
              const MultiplicativeOperator & mul = MultiplicativeOperator(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator<
MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< Input↩
Type1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value
&&!std::is_same< InputType2, void >::value, void >::type * const  = nullptr )
```

Right-handed in-place masked sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.

See the documentation of grb::mxv for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.5 mxv() [4/6]

```
RC grb::mxv (
              Vector< IOType, backend, Coords > & u,
              const Vector< InputType3, backend, Coords > & u_mask,
              const Matrix< InputType2, backend, RIT, CIT, NIT > & A,
              const Vector< InputType1, backend, Coords > & v,
              const Vector< InputType4, backend, Coords > & v_mask,
              const Semiring & semiring = Semiring(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< grb::is_semiring< Semiring >::value &&!grb::is_object<
IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >↩
::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value, void
>::type * const  = nullptr )
```

Right-handed in-place doubly-masked sparse matrix times vector multiplication, $u = u + Av$.

Aliases to this function exist that do not include masks:

- grb::mxv(u, u_mask, A, v, semiring);
- grb::mxv(u, A, v, semiring); When masks are omitted, the semantics shall be the same as though a dense Boolean vector of the appropriate size with all elements set to `true` was given as a mask. We thus describe the semantics of the fully masked variant only.

Note

If only an input mask *v_mask* is intended to be given (and no output mask *u_mask*), then *u_mask* must nonetheless be explicitly given. Passing an empty Boolean vector for *u_mask* is sufficient.

Let u, u_mask be vectors of size m, let v, v_mask be vectors of size n, and let A be an $m \times n$ matrix. Then, a call to this function computes $u = u + Av$ but:

1. only for the elements u_i for which u_mask_i evaluates `true`; and
2. only considering the elements v_j for which v_mask_v evaluates `true`, and otherwise substituting the zero element under the given semiring.

When multiplying a matrix nonzero element $a_{ij} \in A$, it shall be multiplied with an element x_j using the multiplicative operator of the given *semiring*.

When accumulating multiple contributions of multiplications of nonzeroes on some row i, the additive operator of the given *semiring* shall be used.

Nonzero resulting from computing Av are accumulated into any pre- existing values in u by the additive operator of the given *semiring*.

If elements from v, A, or u were missing, the zero identity of the given *semiring* is substituted.

If nonzero values from A were missing, the one identity of the given semiring is substituted.

Note

A nonzero in A may not have a nonzero value in case it is declared as `grb::Matrix< void >`.

The following template arguments *may* be explicitly given:

Template Parameters

descr	Any combination of one or more grb::descriptors. When ommitted, the default grb::descriptors:no_operation will be ass
Semiring	The generalised semiring the matrix–vector multiplication is to be executed under.

The following template arguments will be inferred from the input arguments:

Template Parameters

IOType	The type of the elements of the output vector *u*.
InputType1	The type of the elements of the input vector *v*.
InputType2	The type of the elements of the input matrix *A*.
InputType3	The type of the output mask (*u_mask*) elements.
InputType4	The type of the input mask (*v_mask*) elements.

The following arguments are mandatory:

Parameters

`in,out`	*u*	The output vector.
`in`	*A*	The input matrix. Its grb::nrows must equal the grb::size of *u*.
`in`	*v*	The input vector. Its grb::size must equal the grb::ncols of *A*.

Parameters

`in`	*semiring*	The semiring to perform the matrix–vector multiplication under. Unless grb::descriptors::no_casting is defined, elements from *u*, *A*, and *v* will be cast to the domains of the additive and multiplicative operators of *semiring*.

The vector *v* may not be the same as *u*.

Instead of passing a *semiring*, users may opt to provide an additive commutative monoid and a binary multiplicative operator instead. In this case, *A* may not be a pattern matrix (that is, it must not be of type `grb::Matrix< void >`).

The *semiring* (or the commutative monoid - binary operator pair) is optional if they are passed as a template argument instead.

Note

When providing a commutative monoid - binary operator pair, ALP backends are precluded from employing distributative laws in generating optimised codes.

Non-mandatory arguments are:

Parameters

in	*u_mask*	The output mask. The vector must be of equal size as *u*, *or* it must be empty (have size zero).
in	*v_mask*	The input mask. The vector must be of equal size as *v*, *or* it must be empty (have size zero).
in	*phase*	The requested phase for this primitive—see grb::Phase for details.

The vectors *u_mask* and *v_mask* may never be the same as *u*.

An empty *u_mask* will behave semantically the same as providing no mask; i.e., as a mask that evaluates `true` at every position.

If *phase* is not given, it will be set to the default grb::EXECUTE.

If *phase* is grb::EXECUTE, then the capacity of *u* must be greater than or equal to the capacity required to hold all output elements of the requested computation.

The above semantics may be changed by the following descriptors:

- descriptors::transpose_matrix: A is interpreted as A^T instead.

- descriptors::add_identity: the matrix A is instead interpreted as $A + \mathbf{1}$, where $\mathbf{1}$ is the one identity (i.e., multiplicative identity) of the given *semiring*.
- descriptors::invert_mask: u_i will be written to if and only if u_mask_i evaluates `false`, and v_j will be read from if and only if v_mask_j evaluates `false`.
- descriptors::structural: when evaluating $mask_i$, only the structure of u_mask, v_mask is considered, as opposed to considering their values.
- descriptors::structural_complement: a combination of two descriptors: descriptors::structural and descriptors::invert_mask.
- descriptors::use_index: when reading v_i, then, if there is indeed a nonzero v_i, use the value i instead. This casts the index from `size_t` to the *InputType1* of *v*.
- descriptors::explicit_zero: if u_i was unassigned on entry and if $(Av)_i$ is $\mathbf{0}$, then instead of leaving u_i unassigned, it is set to $\mathbf{0}$ explicitly. Here, $\mathbf{0}$ is the additive identity of the provided *semiring*.
- descriptors::safe_overlap: the vectors *u* and *v* may now be the same container. The user guarantees that no race conditions exist during the requested computation, however. The user may guarantee this due to a a very specific structure of *A* and *v*, or via an intelligently constructed *u_mask*, for example.

Returns

grb::SUCCESS If the computation completed successfully.

grb::MISMATCH If there is at least one mismatch between vector dimensions or between vectors and the given matrix.

grb::OVERLAP If two or more provided vectors refer to the same container while this was not allowed.

When any of the above non-SUCCESS error code is returned, it shall be as though the call was never made– the state of all container arguments and of the application remain unchanged, save for the returned error code.

Returns

grb::PANIC Indicates that the application has entered an undefined state.

Note

Should this error code be returned, the only sensible thing to do is exit the application as soon as possible, while refraining from using any other ALP pritimives.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.6 mxv() [5/6]

```
RC grb::mxv (
             Vector< IOType, implementation, Coords > & u,
             const Matrix< InputType2, implementation, RIT, CIT, NIT > & A,
             const Vector< InputType1, implementation, Coords > & v,
             const Ring & ring,
             typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *  =
nullptr )
```

Right-handed in-place sparse matrix–vector multiplication, $u = u + Av$, over a given semiring.

See the documentation of grb::mxv for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.7 mxv() [6/6]

```
RC grb::mxv (
             Vector< IOType, implementation, Coords > & u,
             const Vector< InputType3, implementation, Coords > & mask,
             const Matrix< InputType2, implementation, RIT, CIT, NIT > & A,
             const Vector< InputType1, implementation, Coords > & v,
             const Ring & ring = Ring(),
             const Phase & phase = EXECUTE,
             typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *  =
nullptr )
```

Right-handed in-place masked sparse matrix–vector multiplication, $u = u + Av$, over a given semiring.

See the documentation of grb::mxv for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.8 vxm() [1/6]

```
RC grb::vxm (
              Vector< IOType, backend, Coords > & u,
              const Vector< InputType1, backend, Coords > & v,
              const Matrix< InputType2, backend, RIT, CIT, NIT > & A,
              const AdditiveMonoid & add = AdditiveMonoid(),
              const MultiplicativeOperator & mul = MultiplicativeOperator(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator<
MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< Input↩
Type1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >↩
::value, void >::type * const  = nullptr )
```

Left-handed in-place sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

See the documentation of grb::vxm for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.9 vxm() [2/6]

```
RC grb::vxm (
              Vector< IOType, backend, Coords > & u,
              const Vector< InputType3, backend, Coords > & mask,
              const Vector< InputType1, backend, Coords > & v,
              const Vector< InputType4, backend, Coords > & v_mask,
              const Matrix< InputType2, backend, RIT, CIT, NIT > & A,
              const AdditiveMonoid & add = AdditiveMonoid(),
              const MultiplicativeOperator & mul = MultiplicativeOperator(),
              const Phase & phase = EXECUTE,
              const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator<
MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< Input↩
Type1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value
&&!grb::is_object< InputType4 >::value &&!std::is_same< InputType2, void >::value, void >↩
::type * const  = nullptr )
```

Left-handed in-place doubly-masked sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

See the documentation of grb::vxm for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.10 vxm() [3/6]

```
RC grb::vxm (
              Vector< IOType, backend, Coords > & u,
              const Vector< InputType3, backend, Coords > & u_mask,
              const Vector< InputType1, backend, Coords > & v,
              const Vector< InputType4, backend, Coords > & v_mask,
              const Matrix< InputType2, backend, RIT, CIT, NIT > & A,
              const Semiring & semiring = Semiring(),
              const Phase & phase = EXECUTE,
              typename std::enable_if< grb::is_semiring< Semiring >::value &&!grb::is_object<
InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >↩
::value &&!grb::is_object< InputType4 >::value &&!grb::is_object< IOType >::value, void >↩
::type *  = nullptr )
```

Left-handed in-place doubly-masked sparse matrix times vector multiplication, $u = u + vA$.

A call to this function is exactly equivalent to calling

- grb::vxm(u, u_mask, A, v, v_mask, semiring, phase) with the descriptors::transpose_matrix flipped.

See the documentation of grb::mxv for the full semantics of this function. Like with grb::mxv, aliases to this function exist that do not include masks:

- grb::vxm(u, u_mask, v, A, semiring, phase);
- grb::vxm(u, v, A, semiring, phase);

Similarly, aliases to this function exist that take an additive commutative monoid and a multiplicative binary operator instead of a semiring.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.11 vxm() [4/6]

```
RC grb::vxm (
              Vector< IOType, implementation, Coords > & u,
              const Vector< InputType1, implementation, Coords > & v,
              const Matrix< InputType2, implementation, RIT, CIT, NIT > & A,
              const Ring & ring = Ring(),
              const Phase & phase = EXECUTE,
              typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *  =
nullptr )
```

Left-handed in-place sparse matrix–vector multiplication, $u = u + vA$, over a given semiring.

See the documentation of grb::vxm for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.12 vxm() [5/6]

```
RC grb::vxm (
              Vector< IOType, implementation, Coords > & u,
              const Vector< InputType3, implementation, Coords > & mask,
              const Vector< InputType1, implementation, Coords > & v,
              const Matrix< InputType2, implementation, RIT, CIT, NIT > & A,
              const AdditiveMonoid & add = AdditiveMonoid(),
              const MultiplicativeOperator & mul = MultiplicativeOperator(),
              const Phase & phase = EXECUTE,
              typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator<
MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< Input↩
Type1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >↩
::value, void >::type *  = nullptr )
```

Left-handed in-place masked sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

See the documentation of grb::vxm for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.12.2.13 vxm() [6/6]

```
RC grb::vxm (
              Vector< IOType, implementation, Coords > & u,
              const Vector< InputType3, implementation, Coords > & mask,
              const Vector< InputType1, implementation, Coords > & v,
              const Matrix< InputType2, implementation, RIT, CIT, NIT > & A,
              const Ring & ring = Ring(),
              const Phase & phase = EXECUTE,
              typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *  =
nullptr )
```

Left-handed in-place masked sparse matrix–vector multiplication, $u = u + vA$, over a given semiring.

See the documentation of grb::vxm for the full specification of this function.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.13 Level-3 Primitives

A collection of functions that allow GraphBLAS semirings to work on one or more two-dimensional sparse containers (i.e, sparse matrices).

Functions

- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType1 , typename InputType2 , typename CIT , typename RIT , typename NIT , class Semiring , Backend backend>
  RC mxm (Matrix< OutputType, backend, CIT, RIT, NIT > &C, const Matrix< InputType1, backend, CIT, RIT, NIT > &A, const Matrix< InputType2, backend, CIT, RIT, NIT > &B, const Semiring &ring=Semiring(), const Phase &phase=EXECUTE)

  Unmasked and in-place sparse matrix–sparse matrix multiplication (SpMSpM), $C+ = A + B$.
- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , Backend backend, typename Coords >
  RC zip (Matrix< OutputType, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Vector< InputType3, backend, Coords > &z, const Phase &phase=EXECUTE)

  The grb::zip merges three vectors into a matrix.
- template<Descriptor descr = descriptors::no_operation, typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , Backend backend, typename Coords >
  RC zip (Matrix< void, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Phase &phase=EXECUTE)

  Merges two vectors into a `void` *matrix.*

7.13.1 Detailed Description

A collection of functions that allow GraphBLAS semirings to work on one or more two-dimensional sparse containers (i.e, sparse matrices).

7.13.2 Function Documentation

7.13.2.1 mxm()

```
RC grb::mxm (
              Matrix< OutputType, backend, CIT, RIT, NIT > & C,
              const Matrix< InputType1, backend, CIT, RIT, NIT > & A,
              const Matrix< InputType2, backend, CIT, RIT, NIT > & B,
              const Semiring & ring = Semiring(),
              const Phase & phase = EXECUTE )
```

Unmasked and in-place sparse matrix–sparse matrix multiplication (SpMSpM), $C+ = A + B$.

Template Parameters

descr	The descriptors under which to perform the computation. Optional; default is grb::descriptors::no_operation.
OutputType	The type of elements in the output matrix.
InputType1	The type of elements in the left-hand side input matrix.
InputType2	The type of elements in the right-hand side input matrix.
Semiring	The semiring under which to perform the multiplication.

Parameters

in,out	*C*	The matrix into which the multiplication AB is accumulated.
in	*A*	The left-hand side input matrix A.
in	*B*	The left-hand side input matrix B.
in	*ring*	The semiring under which the computation should proceed.
in	*phase*	The grb::Phase the primitive should be executed with. This argument is optional; its default is grb::EXECUTE.

Returns

grb::SUCCESS If the computation completed as intended.

grb::FAILED If the capacity of *C* was insufficient to store the output of multiplying *A* and *B*. If this code is returned, *C* on output appears cleared.

grb::OUTOFMEM If *phase* is grb::RESIZE and an out-of-error condition arose while resizing *C*.

Note

This specification does not account for grb::TRY as that phase is still experimental. See its documentation for details.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.13.2.2 zip() [1/2]

```
RC grb::zip (
              Matrix< OutputType, backend, RIT, CIT, NIT > & A,
              const Vector< InputType1, backend, Coords > & x,
              const Vector< InputType2, backend, Coords > & y,
              const Vector< InputType3, backend, Coords > & z,
              const Phase & phase = EXECUTE )
```

The grb::zip merges three vectors into a matrix.

Interprets three input vectors *x*, *y*, and *z* as a series of row coordinates, column coordinates, and nonzeroes, respectively. The thus-defined nonzeroes of a matrix are then stored in a given output matrix *A*.

The vectors *x*, *y*, and *z* must have equal length, as well as the same number of nonzeroes. If the vectors are sparse, all vectors must have the same sparsity structure.

Note

A variant of this function only takes *x* and *y*, and has that the output matrix *A* has `void` element types.

If this function does not return grb::SUCCESS, the output \ a A will have no contents on function exit.

The matrix *A* must have been pre-allocated to store the nonzero pattern that the three given vectors *x*, *y*, and *z* encode, or otherwise this function returns grb::ILLEGAL.

Note

To ensure that the capacity of *A* is sufficient, a succesful call to grb::resize with grb::nnz of *x* suffices. Alternatively, and with the same effect, a succesful call to this function with *phase* equal to grb::RESIZE instead of grb::SUCCESS suffices also.

Parameters

out	*A*	The output matrix.
in	*x*	A vector of row indices.
in	*y*	A vector of column indices.
in	*z*	A vector of nonzero values.
in	*phase*	The grb::Phase in which the primitive is to proceed. Optional; the default is grb::EXECUTE.

Returns

grb::SUCCESS If *A* was constructed successfully.

grb::MISMATCH If *y* or *z* does not match the size of *x*.

grb::ILLEGAL If *y* or *z* do not have the same number of nonzeroes as *x*.

grb::ILLEGAL If *y* or *z* has a different sparsity pattern from *x*.

grb::FAILED If the capacity of *A* was insufficient to store the given sparsity pattern and *phase* is grb::EXECUTE.

grb::OUTOFMEM If the *phase* is grb::RESIZE and *A* could not be resized to have sufficient capacity to complete this function due to out-of-memory conditions.

Descriptors

None allowed.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.13.2.3 zip() [2/2]

```
RC grb::zip (
            Matrix< void, backend, RIT, CIT, NIT > & A,
            const Vector< InputType1, backend, Coords > & x,
            const Vector< InputType2, backend, Coords > & y,
            const Phase & phase = EXECUTE )
```

Merges two vectors into a `void` matrix.

This is a specialisation of grb::zip for pattern matrices. The two input vectors *x* and *y* represent coordinates of nonzeroes to be stored in *A*.

Performance semantics

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

7.14 Nonblocking backend configuration

All configuration parameters for the grb::nonblocking backend.

Namespaces

- namespace grb::config

  Compile-time configuration constants as well as implementation details that are derived from such settings.

Classes

- class ANALYTIC_MODEL

  Configuration parameters relating to the analytic model employed by the nonblocking backend.
- class IMPLEMENTATION< nonblocking >

  Implementation-dependent configuration parameters for the nonblocking backend.
- class PIPELINE

  Configuration parameters relating to the pipeline data structure.

7.14.1 Detailed Description

All configuration parameters for the grb::nonblocking backend.

7.15 Performance Semantics

Each ALP primitive, every constructor, and every destructor come with *performance semantics*, in addition to functional semantics.

Each ALP primitive, every constructor, and every destructor come with *performance semantics*, in addition to functional semantics.

Performance semantics may differ for different backends– ALP stringently mandates that backends defines them, thus imposing a significant degree of predictability on implementations of ALP, but does not significantly limit possible implementation choices.

Warning

> Performance semantics should not be mistaken for performance *guarantees*. The vast majority of computing platforms exhibit performance variabilities that preclude defining stringent such guarantees.

Performance semantics includes classical asymptotic work analysis in the style of Cormen et alii, as commonly taught as part of basic computer science courses. Aside from making the reasonable (although arguably too uncommon) demand that ALP libraries must clearly document the work complexity of the primitives it defines, ALP furthermore demands such analyses for the following quantities:

- how many times operator(s) may be applied,
- intra-process data movement from main memory to processing units,
- new dynamic memory allocations and/or releases of previously allocated memory, and
- whether system calls may occur during a call to the given primitive.

Note

> Typically (but not always) the amount of work is proportional to the number of operator applications.
>
> Typically (but not necessarily always) if primitives are allowed to allocate or free dynamic memory, then it may also thus make system calls.

For backends that allow for more than one user process, the following additional performance semantics must be defined:

- inter-process data movement, and
- how many synchronisation steps a primitive requires to complete.

Defining such performance semantics are crucial to

1. allow algorithm designers to design the best possible algorithms even if the target platforms and target use cases vary,
2. allow users to determine scalability under increasing problem sizes, and
3. allow system architects to determine the qualitative effect of scaling up system resources in an a-priori fashion.

These advantages furthermore do not require expensive experimentation on the part of algorithm designers, users, or system architects. However, it puts a significant demand on the implementers and maintainers of ALP.

See also

> Backends

Author

> A. N. Yzelman, Huawei Technologies Switzerland AG (2020-current)

7.16 Reference and reference_omp backend configuration

All configuration parameters for the grb::reference and the grb::reference_omp backends.

Classes

- class IMPLEMENTATION< reference >

  This class collects configuration parameters that are specific to the grb::reference backend.
- class IMPLEMENTATION< reference_omp >

  This class collects configuration parameters that are specific to the grb::reference_omp backend.
- class PREFETCHING< backend >

  Default prefetching settings for reference and reference_omp backends.

Enumerations

- enum ALLOC_MODE { ALIGNED , INTERLEAVED }

  The memory allocation modes implemented in the grb::reference and the grb::reference_omp backends.

Functions

- std::string **toString** (const ALLOC_MODE mode)

  Converts instances of grb::config::ALLOC_MODE to a descriptive lower-case string.

7.16.1 Detailed Description

All configuration parameters for the grb::reference and the grb::reference_omp backends.

7.16.2 Enumeration Type Documentation

7.16.2.1 ALLOC_MODE

`enum ALLOC_MODE`

The memory allocation modes implemented in the grb::reference and the grb::reference_omp backends.

Enumerator

ALIGNED	Allocation via `posix_memalign`.
INTERLEAVED	Allocation via `numa_alloc_interleaved`.

Chapter 8

Namespace Documentation

8.1 grb Namespace Reference

The ALP/GraphBLAS namespace.

Namespaces

- namespace algorithms

  The namespace for ALP/GraphBLAS algorithms.
- namespace config

  Compile-time configuration constants as well as implementation details that are derived from such settings.
- namespace descriptors

  Collection of standard descriptors.
- namespace identities

  Standard identities common to many operators.
- namespace interfaces

  The namespace for programming APIs that automatically translate to ALP/GraphBLAS.
- namespace operators

  This namespace holds various standard operators such as grb::operators::add and grb::operators::mul.

Classes

- class Benchmarker

  A class that follows the API of the grb::Launcher, but instead of launching the given ALP program once, it launches it multiple times while benchmarking its execution times.
- class collectives

  A static class defining various collective operations on scalars.
- struct has_immutable_nonzeroes

  Used to inspect whether a given semiring has immutable nonzeroes under addition.
- struct is_associative

  Used to inspect whether a given operator or monoid is associative.
- struct is_commutative

  Used to inspect whether a given operator or monoid is commutative.
- struct is_container

Used to inspect whether a given type is an ALP/GraphBLAS container.

- struct is_idempotent

  Used to inspect whether a given operator or monoid is idempotent.
- struct is_monoid

  Used to inspect whether a given type is an ALP monoid.
- struct is_object

  Used to inspect whether a given type is an ALP/GraphBLAS object.
- struct is_operator

  Used to inspect whether a given type is an ALP operator.
- struct is_semiring

  Used to inspect whether a given type is an ALP semiring.
- class Launcher

  A group of user processes that together execute ALP programs.
- class Matrix

  An ALP/GraphBLAS matrix.
- class Monoid

  A generalised monoid.
- class PinnedVector

  Provides a mechanism to access ALP containers from outside of an ALP context.
- class Properties

  Collection of various properties on the given ALP/GraphBLAS backend.
- class Semiring

  A generalised semiring.
- class spmd

  For backends that support multiple user processes this class defines some basic primitives to support SPMD programming.
- class Vector

  A GraphBLAS vector.

Typedefs

- typedef unsigned int Descriptor

  Descriptors indicate pre- or post-processing for some or all of the arguments to an ALP/GraphBLAS call.

Enumerations

- enum Backend {
  reference , reference_omp , hyperdags , nonblocking ,
  shmem1D , **NUMA1D** , **GENERIC_BSP** , BSP1D ,
  doublyBSP1D , **BSP2D** , **autoBSP** , **optBSP** ,
  hybrid , **hybridSmall** , **hybridMid** , **hybridLarge** ,
  minFootprint , banshee , **banshee_ssr** }

  A collection of all backends.
- enum EXEC_MODE { AUTOMATIC = 0 , MANUAL , FROM_MPI }

  The various ways in which the grb::Launcher can be used to execute an ALP program.
- enum IOMode { SEQUENTIAL = 0 , PARALLEL }

  The GraphBLAS input and output functionalities can either be used in a sequential or parallel fashion.
- enum Phase { RESIZE , TRY , EXECUTE }

  Primitives with sparse ALP/GraphBLAS output containers may run into the issue where an appropriate grb::capacity may not always be clear.
- enum RC {
  SUCCESS = 0 , PANIC , OUTOFMEM , MISMATCH ,
  OVERLAP , OVERFLW , UNSUPPORTED , ILLEGAL ,
  FAILED }

  Return codes of ALP primitives.

Functions

- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType1 , typename InputType2 , typename OutputType >
  static enum RC apply (OutputType &out, const InputType1 &x, const InputType2 &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< OutputType >::value, void >::type *=nullptr)

  Out-of-place application of the operator OP on two data elements.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename fwd_iterator3 = const InputType * __restrict__, Backend implementation = config::default_backend>
  RC buildMatrixUnique (Matrix< InputType, implementation > &A, fwd_iterator1 I, const fwd_iterator1 I_end, fwd_iterator2 J, const fwd_iterator2 J_end, fwd_iterator3 V, const fwd_iterator3 V_end, const IOMode mode)

  Assigns nonzeroes to the matrix from a coordinate format.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename fwd_iterator3 = const InputType * __restrict__, Backend implementation = config::default_backend>
  RC **buildMatrixUnique** (Matrix< InputType, implementation > &A, fwd_iterator1 I, fwd_iterator2 J, fwd_↩ iterator3 V, const size_t nz, const IOMode mode)

  Alias that transforms a set of pointers and an array length to the buildMatrixUnique variant based on iterators.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename RIT , typename CIT , typename NIT , typename fwd_iterator , Backend implementation = config::default_backend>
  RC buildMatrixUnique (Matrix< InputType, implementation, RIT, CIT, NIT > &A, fwd_iterator start, const fwd_iterator end, const IOMode mode)

  Version of buildMatrixUnique that works by supplying a single iterator (instead of three).
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename RIT , typename CIT , typename NIT , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename length_type = size_↩ t, Backend implementation = config::default_backend>
  RC **buildMatrixUnique** (Matrix< InputType, implementation, RIT, CIT, NIT > &A, fwd_iterator1 I, fwd_↩ iterator2 J, const length_type nz, const IOMode mode)

  Version of the above buildMatrixUnique that handles nullptr value pointers.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator , Backend backend, typename Coords >
  RC buildVector (Vector< InputType, backend, Coords > &x, fwd_iterator start, const fwd_iterator end, const IOMode mode)

  Constructs a dense vector from a container of exactly grb::size(x) elements.
- template<Descriptor descr = descriptors::no_operation, typename InputType , class Merger = operators::right_assign< InputType >, typename fwd_iterator1 , typename fwd_iterator2 , Backend backend, typename Coords >
  RC buildVector (Vector< InputType, backend, Coords > &x, fwd_iterator1 ind_start, const fwd_↩ iterator1 ind_end, fwd_iterator2 val_start, const fwd_iterator2 val_end, const IOMode mode, const Merger &merger=Merger())

  Ingests possibly sparse input from a container to which iterators are provided.
- template<Descriptor descr = descriptors::no_operation, typename InputType , class Merger = operators::right_assign< InputType >, typename fwd_iterator1 , typename fwd_iterator2 , Backend backend, typename Coords >
  RC buildVectorUnique (Vector< InputType, backend, Coords > &x, fwd_iterator1 ind_start, const fwd_↩ iterator1 ind_end, fwd_iterator2 val_start, const fwd_iterator2 val_end, const IOMode mode)

  Ingests a set of nonzeroes into a given vector x.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t capacity (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Queries the capacity of the given ALP/GraphBLAS container.
- template<typename InputType , Backend backend, typename Coords >
  size_t capacity (const Vector< InputType, backend, Coords > &x) noexcept

  Queries the capacity of the given ALP/GraphBLAS container.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  RC clear (Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

Clears a given matrix of all nonzeroes.

- template<typename DataType , Backend backend, typename Coords >
  RC clear (Vector< DataType, backend, Coords > &x) noexcept

  Clears a given vector of all nonzeroes.

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , Backend backend, typename Coords >
  RC dot (IOType &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_↩
  if< !grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< IOType >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the dot product, $z+ = (x, y)$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class AddMonoid , class AnyOp , typename OutputType , typename InputType1 , typename InputType2 , enum Backend backend, typename Coords >
  RC dot (OutputType &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const AddMonoid &addMonoid=AddMonoid(), const AnyOp &anyOp=AnyOp(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Add↩
  Monoid >::value &&grb::is_operator< AnyOp >::value, void >::type *const =nullptr)

  Calculates the dot product, $z+ = (x, y)$, under a given additive monoid and multiplicative operator.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename Input↩
  Type2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Input↩
  Type2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< Input↩
  Type2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + \beta$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename Input↩
  Type2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + y$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename Input↩
  Type2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = x. + \beta$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition of two vectors, $z+ = x. + y$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + \beta$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+=\alpha.+y$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+=x.+\beta$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition of two vectors, $z+=x.+y$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z=\alpha\odot\beta$, out of place, monoid version.
- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z=\alpha\odot\beta$, out of place, operator version.
- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z=\alpha\odot y$, out of place, monoid version.
- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z=\alpha\odot y$, out of place, operator version.
- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE,

const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Computes $z = x \odot \beta$, out of place, monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = x \odot \beta$, out of place, operator variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = x \odot y$, out of place, monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = x \odot y$, out of place, operator variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot \beta$, out of place, masked monoid version.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot \beta$, out of place, operator and masked version.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot y$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType

>::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Computes $z = \alpha \odot y$, out of place, masked operator version.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Computes $z = x \odot \beta$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Computes $z = x \odot \beta$, out of place, masked operator variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Computes $z = x \odot y$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Computes $z = x \odot y$, out of place, masked operator variant.

- template<typename Func , typename DataType , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_backend, typename... Args>
RC eWiseLambda (const Func f, const Matrix< DataType, implementation, RIT, CIT, NIT > &A, Args...)

Executes an arbitrary element-wise user-defined function f on all nonzero elements of a given matrix A.

- template<typename Func , typename DataType , Backend backend, typename Coords , typename... Args>
RC eWiseLambda (const Func f, const Vector< DataType, backend, Coords > &x, Args...)

Executes an arbitrary element-wise user-defined function f on any number of vectors of equal length.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
RC eWiseMul (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

In-place element-wise multiplication of two scalars, $z+ = \alpha. * \beta$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
RC eWiseMul (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

In-place element-wise multiplication of a scalar and vector, $z+ = \alpha . * y$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a vector and scalar, $z+ = x . * \beta$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of two vectors, $z+ = x . * y$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of two scalars, $z+ = \alpha . * \beta$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a scalar and vector, $z+ = \alpha . * y$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a vector and scalar, $z+ = x . * \beta$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of two vectors, $z+ = x . * y$, under a given semiring, masked variant.

- template<enum Backend backend = config::default_backend>
  RC finalize ()

  Finalises an ALP/GraphBLAS context opened by the last call to grb::init.

- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType , typename IOType >
  static RC foldl (IOType &x, const InputType &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Application of the operator OP on two data elements.

- template<Descriptor descr = descriptors::no_operation, class Monoid , typename IOType , typename InputType , Backend backend, typename Coords >
  RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Folds a vector into a scalar, left-to-right.
- template<Descriptor descr = descriptors::no_operation, class Monoid , typename InputType , typename IOType , typename MaskType , Backend backend, typename Coords >
  RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Vector< MaskType, backend, Coords > &mask, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< InputType >::value &&!grb::is_object< MaskType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Reduces, or folds, a vector into a scalar.
- template<Descriptor descr = descriptors::no_operation, class OP , typename IOType , typename InputType , typename MaskType , Backend backend, typename Coords >
  RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Vector< MaskType, backend, Coords > &mask, const OP &op=OP(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< MaskType >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Folds a vector into a scalar, left-to-right.
- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType , typename IOType >
  static RC foldr (const InputType &x, IOType &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Application of the operator OP on two data elements.
- template<Descriptor descr = descriptors::no_operation, class Monoid , typename InputType , typename IOType , typename MaskType , Backend backend, typename Coords >
  RC foldr (const Vector< InputType, backend, Coords > &x, const Vector< MaskType, backend, Coords > &mask, IOType &y, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< InputType >::value &&!grb::is_object< MaskType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Folds a vector into a scalar, right-to-left.
- template<Descriptor descr = descriptors::no_operation, class Monoid , typename IOType , typename InputType , Backend backend, typename Coords >
  RC foldr (const Vector< InputType, backend, Coords > &y, IOType &x, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Folds a vector into a scalar, right-to-left.
- template<typename ElementType , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_backend>
  uintptr_t getID (const Matrix< ElementType, implementation, RIT, CIT, NIT > &x)

  Specialisation of getID for matrix containers.
- template<typename ElementType , typename Coords , Backend implementation = config::default_backend>
  uintptr_t getID (const Vector< ElementType, implementation, Coords > &x)

  Function that returns a unique ID for a given non-empty container.
- template<enum Backend backend = config::default_backend>
  RC init ()

  Initialises the calling user process.
- template<enum Backend backend = config::default_backend>
  RC init (const size_t s, const size_t P, void *const implementation_data)

  Initialises the calling user process.
- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType1 , typename InputType2 , typename CIT , typename RIT , typename NIT , class Semiring , Backend backend>
  RC mxm (Matrix< OutputType, backend, CIT, RIT, NIT > &C, const Matrix< InputType1, backend, CIT, RIT, NIT > &A, const Matrix< InputType2, backend, CIT, RIT, NIT > &B, const Semiring &ring=Semiring(), const Phase &phase=EXECUTE)

Unmasked and in-place sparse matrix–sparse matrix multiplication (SpMSpM), $C += A + B$.

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place sparse matrix–vector multiplication, $u = u + Av$*, over a given commutative additive monoid and any binary operator acting as multiplication.*

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place doubly-masked sparse matrix–vector multiplication, $u = u + Av$*, over a given commutative additive monoid and any binary operator acting as multiplication.*

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Matrix< InputType2, backend, RIT, NIT, CIT > &A, const Vector< InputType1, backend, Coords > &v, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place masked sparse matrix–vector multiplication, $u = u + Av$*, over a given commutative additive monoid and any binary operator acting as multiplication.*

- template<Descriptor descr = descriptors::no_operation, class Semiring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &u_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Semiring &semiring=Semiring(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_semiring< Semiring >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value, void >::type *const =nullptr)

  Right-handed in-place doubly-masked sparse matrix times vector multiplication, $u = u + Av$.

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_backend>
  RC mxv (Vector< IOType, implementation, Coords > &u, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Vector< InputType1, implementation, Coords > &v, const Ring &ring, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Right-handed in-place sparse matrix–vector multiplication, $u = u + Av$*, over a given semiring.*

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , typename Coords , enum Backend implementation = config::default_backend>

RC mxv (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords > &mask, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Vector< InputType1, implementation, Coords > &v, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std↩ ::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

Right-handed in-place masked sparse matrix–vector multiplication, $u = u + Av$, over a given semiring.

- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
size_t ncols (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Requests the column size of a given matrix.

- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
size_t nnz (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Retrieve the number of nonzeroes contained in this matrix.

- template<typename DataType , Backend backend, typename Coords >
size_t nnz (const Vector< DataType, backend, Coords > &x) noexcept

  Request the number of nonzeroes in a given vector.

- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
size_t nrows (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Requests the row size of a given matrix.

- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
RC resize (Matrix< InputType, backend, RIT, CIT, NIT > &A, const size_t new_nz) noexcept

  Resizes the nonzero capacity of this matrix.

- template<typename InputType , Backend backend, typename Coords >
RC resize (Vector< InputType, backend, Coords > &x, const size_t new_nz) noexcept

  Resizes the nonzero capacity of this vector.

- template<Descriptor descr = descriptors::no_operation, typename DataType , typename T , typename Coords , Backend backend>
RC set (Vector< DataType, backend, Coords > &x, const T val, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr) noexcept

  Sets all elements of a vector to the given value.

- template<Descriptor descr = descriptors::no_operation, typename DataType , typename MaskType , typename T , Backend backend, typename Coords >
RC set (Vector< DataType, reference, Coords > &x, const Vector< MaskType, backend, Coords > &mask, const T val, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr)

  Sets all elements of a vector to the given value whenever the given mask evaluates `true`.

- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType , Backend backend, typename Coords >
RC set (Vector< OutputType, backend, Coords > &x, const Vector< InputType, backend, Coords > &y, const Phase &phase=EXECUTE)

  Sets the content of a given vector x to be equal to that of another given vector y.

- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename MaskType , typename InputType , Backend backend, typename Coords >
RC set (Vector< OutputType, backend, Coords > &x, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType, backend, Coords > &y, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType >::value, void >::type *const =nullptr)

  Sets the content of a given vector x to be equal to that of another given vector y.

- template<Descriptor descr = descriptors::no_operation, typename DataType , typename T , Backend backend, typename Coords >
RC setElement (Vector< DataType, backend, Coords > &x, const T val, const size_t i, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr)

  Sets the element of a given vector at a given position to a given value.

- template<typename DataType , Backend backend, typename Coords >
size_t size (const Vector< DataType, backend, Coords > &x) noexcept

  Request the size of a given vector.

- std::string toString (const RC code)
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType1, backend, Coords > &v, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Left-handed in-place sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Left-handed in-place doubly-masked sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.
- template<Descriptor descr = descriptors::no_operation, class Semiring , typename IOType , typename InputType1 , typename Input↩ Type2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &u_mask, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Semiring &semiring=Semiring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Semiring >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Left-handed in-place doubly-masked sparse matrix times vector multiplication, $u = u + vA$.
- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend implementation = config::default_backend>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Left-handed in-place sparse matrix–vector multiplication, $u = u + vA$, over a given semiring.
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , Backend implementation>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords > &mask, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *=nullptr)

  Left-handed in-place masked sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.
- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend implementation = config↩ ::default_backend>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords

> &mask, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std::enable↩ _if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

Left-handed in-place masked sparse matrix–vector multiplication, $u = u + vA$, over a given semiring.

- template<Backend backend = config::default_backend>
  RC wait ()

  Depending on the backend, ALP/GraphBLAS primitives may be non-blocking, meaning that the operation immediately returns even though the requested computation has not been performed.
- template<Backend backend, typename InputType , typename RIT , typename CIT , typename NIT , typename... Args>
  RC wait (const Matrix< InputType, backend, RIT, CIT, NIT > &A, const Args &... args)

  A variant of grb::wait that executes, at minimum, all nonblocking primitives required for computing a given output matrix as well as, optionally, for any additional output containers given in the variadic argument list.
- template<Backend backend, typename InputType , typename Coords , typename... Args>
  RC wait (const Vector< InputType, backend, Coords > &x, const Args &... args)

  A variant of grb::wait that executes, at minimum, all nonblocking primitives required for computing a given output vector as well as, optionally, for any additional output containers given in the variadic argument list.
- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , Backend backend, typename Coords >
  RC zip (Matrix< OutputType, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Vector< InputType3, backend, Coords > &z, const Phase &phase=EXECUTE)

  The grb::zip merges three vectors into a matrix.
- template<Descriptor descr = descriptors::no_operation, typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , Backend backend, typename Coords >
  RC zip (Matrix< void, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Phase &phase=EXECUTE)

  Merges two vectors into a `void` matrix.

8.1.1 Detailed Description

The ALP/GraphBLAS namespace.

All ALP/GraphBLAS primitives, container types, algebraic structures, and type traits are defined within.

8.1.2 Typedef Documentation

8.1.2.1 Descriptor

```
typedef unsigned int Descriptor
```

Descriptors indicate pre- or post-processing for some or all of the arguments to an ALP/GraphBLAS call.

An example is to transpose the input matrix during a sparse matrix–vector multiplication: `grb::mxv< grb↩ ::descriptors::transpose_matrix >( y, A, x, ring );` the above thus computes $y \to y + A^T x$ and not $y \to y + Ax$.

Such pre-processing often happens on-the-fly, without significant overhead to the primitive costings in any of its cost dimensions – work, intra- and inter-process data movement, synchronisations, and memory usage.

Note

If the application of a descriptor is *not* without significant overhead, a backend *must* clearly indicate so.

Descriptors may be combined using bit-wise operators. For instance, to both indicate the matrix needs be transposed and the mask needs be inverted, the following descriptor can be passed: `transpose_matrix | invert_mask`

8.1.3 Enumeration Type Documentation

8.1.3.1 EXEC_MODE

`enum EXEC_MODE`

The various ways in which the grb::Launcher can be used to execute an ALP program.

Warning

An implementation may require different linker commands when using different modes.

Depending on the mode given to grb::Launcher, the parameters required for the exec function may differ.

Note

However, the ALP program is unaware of which mode is the launcher employs and will not have to change.

Enumerator

AUTOMATIC	Automatic mode. The grb::Launcher can spawn user processes which will execute a given program.

Enumerator

MANUAL	Manual mode. The user controls *nprocs* user processes which together should execute a given program, by, for example, using the grb::Launcher.
FROM_MPI	When running from an MPI program. The user controls *nprocs* MPI programs, which, together, should execute a given ALP program.

8.1.3.2 IOMode

`enum IOMode`

The GraphBLAS input and output functionalities can either be used in a sequential or parallel fashion.

Input functions such as buildVector or buildMatrixUnique default to sequential behaviour, which means that the collective calls to either function must have the exact same arguments– that is, each user process is passed the exact same input data.

Note

This does not necessarily mean that all data is stored in a replicated fashion across all user processes.

This default behaviour comes with obvious performance penalties; each user process must scan the full input data set, which takes $\Theta(n)$ time. Scalable behaviour would instead incur $\Theta(n/P)$ time, with *P* the number of user processes. Using a parallel IOMode provides exactly this scalable performance. On input, this means that each user process can pass different data to the same collective call to, e.g., buildVector or buildMatrixUnique.

For output, which GraphBLAS provides via *const* iterators, sequential mode means that each user process retrieves an iterator over all output elements– this requires costly all-to-all communication. Parallel mode output instead only returns those elements that do not require inter user- process communication.

Note

It is guaranteed the union of all output over all user processes corresponds to all elements in the GraphBLAS container.

See the respective functions and classes for full details:

1. grb::buildVector;
2. grb::buildMatrixUnique;
3. grb::Vector::const_iterator;
4. grb::Matrix::const_iterator.

Enumerator

SEQUENTIAL	Sequential mode IO. Use of this mode results in non-scalable input and output. Its use is recommended only in case of small data sets or in one-off situations.

Enumerator

PARALLEL	Parallel mode IO. Use of this mode results in fully scalable input and output. Its use is recommended as a default. Note that this does require the user to have his or her data distributed over the various user processes on input, and requires the user to handle distributed data on output. This is the default mode on all Graph↩ BLAS IO functions. **Note**

Enumerator

8.1.3.3 Phase

`enum Phase`

Primitives with sparse ALP/GraphBLAS output containers may run into the issue where an appropriate grb::capacity may not always be clear.

This is classically the case for level-3 sparse BLAS primitives, which commonly is solved by splitting up the computation into a symbolic and numeric phase. During the symbolic phase, the computation is simulated in order to derive the required capacity of the output container, which is then immediately resized. Then during the numeric phase, the actual computation is carried out, knowing that the output container is large enough to hold the requested output.

A separation in a symbolic and numeric phase is not the only possible split; for example, required output capacities may be estimated during a first stage, while a second stage will then dynamically allocate additional memory if the estimation proved too optimistic.

We recognise that:

1. not only level-3 primitives may require a two-stage approach– for example, a backend could be designed to support extremely large-sized vectors that contains relatively few nonzeroes, in which case also level-1 and level-2 primitives may benefit of symbolic and numeric phases.
2. especially for level-1 and level-2 primitives, it may also be that single-phase approaches are feasible. Hence ALP/GraphBLAS defines that the execute phase, grb::EXECUTE, is the default when calling an ALP/Graph↩ BLAS primitive without an explicit phase argument.
3. sometimes speculative execution is warranted; these apply to situations where
   (a) capacities are almost surely sufficient, *and*
   (b) partial results, if the full output could not be computed due to capacity issues, are in fact acceptable.

To cater to a wide range of approaches and use cases, we support the following three phases:

1. grb::RESIZE, which resizes capacities based on the requested operation;
2. grb::EXECUTE, which attempts to execute the computation assuming the capacity is sufficient;
3. grb::TRY, which attempts to execute the computation, and does not mind if the capacity turns out to be insufficient.

Backends must give precise performance semantics to primitives executing in each of the three possible phases. Backends can only fail with grb::OUTOFMEM or grb::PANIC when an operation is called using the resize phase and is immediately followed by an equivalent call using the execute phase– otherwise, it must succeed and complete the requested computation.

Summarising the above, a call to any ALP/GraphBLAS primitive f with (potentially sparse) output container A can be made in three ways:

1. f(A, ..., EXECUTE), which shall always be successful if it somehow is guaranteed that *A* has enough capacity prior to the call. If *A* did not have enough capacity, the call to *f* shall fail and the contents of *A*, after function exit, shall be cleared.
2. a successful call to f(A, ..., RESIZE) shall guarantee that a following call to f(A, ..., EXECUTE) is successful;
3. a call to f(A, ..., TRY), which may or may not succeed. If the call does not succeed, then *A*, after function exit:
   (a) contains exactly grb::capacity (of *A*) nonzeroes;
   (b) has nonzeroes at the coordinates where *A* on entry had nonzeroes;
   (c) has nonzeroes with values equal to those that would have been computed at its coordinates were the call successul; and
   (d) does not have computed all nonzeroes that would have been present if the call were successful (or otherwise it should have returned grb::SUCCESS).

Note

Calls can typically also return grb::PANIC, which, if returned, makes undefined the contents of all ALP/↩ GraphBLAS containers as well as makes undefined the state of ALP/GraphBLAS as a whole.

The following code snippets, assuming all unchecked return codes are grb::SUCCESS, thus are semantically equivalent:

```
// default capacity of A is sufficient for \a f to succeed
f( A, ..., EXECUTE );
if( resize( A, sufficient_capacity_for_output_of_f ) == SUCCESS ) {
   f( A, ..., EXECUTE );
}
if( f( A, ..., RESIZE ) == SUCCESS ) {
    f( A, ..., EXECUTE );
}
resize( B, nnz( A ) );
set( B, A );
if( f( A, ..., EXECUTE ) == FAILED ) {
    f( B, ..., RESIZE );
    std::swap( A, B );
}
resize( B, nnz( A ) );
set( B, A );
while( f( A, ..., EXECUTE ) == FAILED ) {
    resize( A, capacity( A ) + 1 );
    set( A, B );
}
```

Note

If the matrix *A* is empty on entry, then the latter two code snippets do not require the use *B* as a temporary buffer.

Since grb::EXECUTE is the default phase, any occurrance of `f( A, ..., EXECUTE )` may be replaced with `f( A, ... )`.

The above code snippets do not include try phases since whenever output containers do not have enough capacity, primitives executed using grb::TRY will *not* generate equivalent results.

Enumerator

Enumerator

RESIZE	Speculatively assumes that the output container(s) of the requested operation lack the necessary capacity to hold all outputs of the computation. Instead of executing the requested operation, this phase attempts to both estimate and resize the output container(s). A successful call using this phase guarantees that a subsequent and equivalent call us

Enumerator

TRY	Speculatively assumes that the output container of the requested operation has enough capacity to complete the computation, and attempts to do so. If the capacity was indeed found to be sufficient, then the computation *must* complete as specified– unless grb::PANIC is returned. If, nevertheless, capacity was not sufficient then the result of the computa-

Enumerator

EXECUTE	Speculatively assumes that the output container of the requested operation has enough capacity to complete the computation, and attempts to do so. If the capacity was indeed found to be sufficient, then the computation *must* complete as specified. In this case, capacities are additionally *not* allowed to be modified by the call to the primitive using

Enumerator

8.1.3.4 RC

`enum RC`

Return codes of ALP primitives.

All primitives that are not *getters* return one of the codes defined here. All primitives may return SUCCESS, and all primitives may return PANIC. All other error codes are optional– please see the description of each primitive which other error codes may be valid.

For core ALP primitives, any non-SUCCESS and non-PANIC error code shall have no side effects; if a call fails, it shall be as though the call was never made.

Enumerator

SUCCESS	Indicates the primitive has executed successfully. All primitives may return this error code.

Enumerator

PANIC	Generic fatal error code. Signals that ALP has entered an undefined state. Users can only do their best to exit their application gracefully once PANIC has been encountered. An implementation (backend) is encouraged to write clear error messages to stderr prior to returning this error code. All primitives may return this error

Enumerator

OUTOFMEM	Signals an out-of-memory error while executing the requested primitive. User can mitigate by freeing memory and retrying the call or by reducing the amount of memory required by this call. This error code may only be returned when explicitly documented as such.

Enumerator

MISMATCH	One or more of the ALP/↩ Graph↩ BLAS objects passed to the primitive that returned this error have mismatching dimensions. User can mitigate by reissuing with correct parameters. It is usually not possible to mitigate at runtime; more often than not, this error signals a logical programming error. This error code may only be returned when

Enumerator

OVERLAP	One or more of the Graph↩ BLAS objects corresponding to the call returning this error refer to the same object while this explicitly is forbidden. **Deprecated** This error code will be replaced with ILLEGAL. User can mitigate by reissuing with correct parameters. It is usually not possible to mitigate at runtime; more often
Generated by Doxygen	than not, this error sig-

Enumerator

OVERFLW	Indicates that execution of the requested primitive with the given arguments would result in overflow. Users can mitigate by modifying the offending call. It is usually not possible to mitigate at runtime; more often than not, this error signals the underlying problem is too large to handle with whatever current resources have been as-

Enumerator

UNSUPPORTED	Indicates that the execution of the requested primitive with the given arguments is not supported by the selected backend. This error code should never be returned by a fully compliant backend. If encountered, the end-user may mitigate by selecting a different backend.

Enumerator

ILLEGAL	A call to a primitive has determined that one of its arguments was illegal as per the specification of the primitive. User can mitigate by reissuing with correct parameters. It is usually not possible to mitigate at runtime; more often than not, this error signals a logical programming error. This error code may only be returned when

Enumerator

FAILED	Indicates when one of the grb::algorithms has failed to achieve its intended result, for instance, when an iterative method failed to converged within its alloted resources. This error code may only be returned when explicitly documented as such, and may never be returned by core ALP primitives– it is reserved for use by algorithms only.

8.1.4 Function Documentation

8.1.4.1 finalize()

```
RC grb::finalize ( )
```

Finalises an ALP/GraphBLAS context opened by the last call to grb::init.

Deprecated Please use grb::Launcher instead. This primitive will be removed from verson 1.0 onwards.

This function must be called collectively and must follow a call to grb::init. After successful execution of this function, a new call to grb::init may be made. (This function is re-entrant.)

After a call to this function, any ALP/GraphBLAS objects that remain in scope become invalid.

Warning

Invalid ALP/GraphBLAS containers will remain invalid no matter if a next call to grb::init is made.

Template Parameters

backend	Which ALP/GraphBLAS backend to finalise.

Returns

SUCCESS If finalisation was successful.

PANIC If this function fails, the state of the ALP/GraphBLAS implementation becomes undefined. This means none of its functions should be called during the remainder program execution; in particular this means a new call to grb::init will not remedy the situation.

Performance semantics

None. Implementations are encouraged to specify the complexity of their implementation of this function in terms of the parameter *P* the matching call to grb::init was called with.

Warning

This primitive has been deprecated since version 0.5. Please update your code to use the grb::Launcher instead.

8.1.4.2 init() [1/2]

```
RC grb::init ( )
```

Initialises the calling user process.

Deprecated Please use grb::Launcher instead. This primitive will be removed from verson 1.0 onwards.

This variant takes no input arguments. It will assume a single user process exists; i.e., the call is equivalent to one to grb::init with *s* zero and *P* one (and *implementation_data* `NULL`).

Template Parameters

backend	The backend implementation to initialise.

Returns

SUCCESS If the initialisation was successful.

PANIC If returned, the state of the ALP library becomes undefined.

Warning

This primitive has been deprecated since version 0.5. Please update your code to use the grb::Launcher instead.

8.1.4.3 init() [2/2]

```
RC grb::init (
              const size_t s,
              const size_t P,
              void *const implementation_data )
```

Initialises the calling user process.

Deprecated Please use grb::Launcher instead. This primitive will be removed from verson 1.0 onwards.

Template Parameters

backend	Which GraphBLAS backend this call to init initialises.

By default, the backend that is selected by the user at compile-time is used. If no backend was selected, grb::reference is assumed.

Parameters

in	*s*	The ID of this user process.
in	*P*	The total number of user processes.

If the backend supports multiple user processes, the user can invoke this function with *P* equal to one or higher; if the backend supports only a single user process, then *P* must equal one.

The value for the user process ID *s* must be larger or equal to zero and must be strictly smaller than *P*. If $P > 1$, each user process must call this function collectively, each user process should pass the same value for *P*, and each user process should pass a unique value for *s* amongst all *P* collective calls made.

Parameters

in	*implementation_data*	Any implementation-defined data structure required for successful completion of this call.

An implementation may define that additional data is required for a call to this function to complete successfully. Such data may be passed via the final argument to this function, *implementation_data*.

If the implementation does not support multiple user processes, then a value for *implementation_data* shall not be required. In particular, a call to this function with an empty parameter list shall then be legal and infer the following default arguments: zero for *s*, one for *P*, and *NULL* for *implementation_data*. When such an implementation is requested to initialise multiple user processes, then grb::UNSUPPORTED shall be returned.

A call to this function must be matched with a call to grb::finalize. After a successful call to this function, a new call to grb::init without first calling grb::finalize shall incur undefined behaviour. The construction of ALP/GraphBLAS containers without a preceding successful call to grb::init will result in undefined behaviour. Any valid GraphBLAS containers will become invalid after a call to grb::finalize.

Returns

SUCCESS If the initialisation was successful.

UNSUPPORTED When the implementation does not support multiple user processes while the given *P* was larger than 1.

PANIC If returned, the state of the ALP library becomes undefined.

After a call to this function that exits with a non-SUCCESS and non-PANIC error code, the program shall behave as though the call were never made.

Note

There is no argument checking. If *s* is larger or equal to *P*, undefined behaviour occurs. If *implementation_data* was invalid or corrupted, undefined behaviour occurs.

Performance semantics

Implementations and backends must specify the complexity of this function in terms of *P*.

Note

Compared to the GraphBLAS C specification, this function lacks a choice whether to execute in 'blocking' or 'non-blocking' mode. With ALP, the selected backend controls whether execution proceeds in a non-blocking manner or not. Thus selecting a blocking backend for compilation results in the application of blocking semantics, while selecting a non-blocking backend results in the application of non- blocking semantics.

Note that in the GraphBLAS C specification, a blocking mode is a valid implementation of a non-blocking mode. Therefore, this specification will still yield a valid C API implementation when properly wrapping around a blocking ALP/GraphBLAS backend.

This specification allows for grb::init to be called multiple times from the same process and the same thread. The parameters *s* and *P* (and *implementation_data*) may differ each time. Each (repeated) call must of course continue to meet all the above requirements.

The GraphBLAS C API does not have the notion of user processes. We believe this notion is necessary to properly integrate into parallel frameworks, and also to affect proper and efficient parallel I/O.

Warning

This primitive has been deprecated since version 0.5. Please update your code to use the grb::Launcher instead.

8.1.4.4 toString()

```
std::string grb::toString (
              const RC code )
```

Returns

A string describing the given error code.

8.2 grb::algorithms Namespace Reference

The namespace for ALP/GraphBLAS algorithms.

Namespaces

- namespace pregel

  The namespace for ALP/Pregel algorithms.

Functions

- template<Descriptor descr = descriptors::no_operation, typename IOType , typename NonzeroType , typename InputType , typename ResidualType , class Semiring = Semiring< operators::add< InputType, InputType, InputType >, operators::mul< IOType, NonzeroType, InputType >, identities::zero, identities::one >, class Minus = operators::subtract< ResidualType >, class Divide = operators::divide< ResidualType >>
RC bicgstab (grb::Vector< IOType > &x, const grb::Matrix< NonzeroType > &A, const grb::Vector< InputType > &b, const size_t max_iterations, ResidualType tol, size_t &iterations, ResidualType &residual, Vector< InputType > &r, Vector< InputType > &rhat, Vector< InputType > &p, Vector< InputType > &v, Vector< InputType > &s, Vector< InputType > &t, const Semiring &semiring=Semiring(), const Minus &minus=Minus(), const Divide &divide=Divide())

  Solves a linear system $b = Ax$ with x unknown by using the bi-conjugate gradient (bi-CG) stabilised method; i.e., BiCGstab.

- template<Descriptor descr = descriptors::no_operation, typename IOType , typename ResidualType , typename NonzeroType , typename InputType , class Ring = Semiring< grb::operators::add< IOType >, grb::operators::mul< IOType >, grb::identities::zero, grb::identities::one >, class Minus = operators::subtract< IOType >, class Divide = operators::divide< IOType >>
grb::RC conjugate_gradient (grb::Vector< IOType > &x, const grb::Matrix< NonzeroType > &A, const grb::Vector< InputType > &b, const size_t max_iterations, ResidualType tol, size_t &iterations, ResidualType &residual, grb::Vector< IOType > &r, grb::Vector< IOType > &u, grb::Vector< IOType > &temp, const Ring &ring=Ring(), const Minus &minus=Minus(), const Divide &divide=Divide())

  Solves a linear system $b = Ax$ with x unknown by the Conjugate Gradients (CG) method on general fields.

- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType1 , typename InputType2 , class Ring , class Division = grb::operators::divide< typename Ring::D3, typename Ring::D3, typename Ring::D4 >>
RC cosine_similarity (OutputType &similarity, const Vector< InputType1 > &x, const Vector< InputType2 > &y, const Ring &ring=Ring(), const Division &div=Division())

  Computes the cosine similarity.

- template<Descriptor descr = descriptors::no_operation, bool criticalSection = false, typename IOType , typename NZType >
RC kcore_decomposition (const Matrix< NZType > &A, Vector< IOType > &core, Vector< IOType > &distances, Vector< IOType > &temp, Vector< IOType > &update, Vector< bool > &status, IOType &k)

  The k-core decomposition algorithm.

- template<Descriptor descr = descriptors::no_operation, typename IOType = double, class Operator = operators::square_diff< IOType, IOType, IOType >>
RC kmeans_iteration (Matrix< IOType > &K, Vector< std::pair< size_t, IOType > > &clusters_and_distances, const Matrix< IOType > &X, const size_t max_iter=1000, const Operator &dist_op=Operator())

  The kmeans iteration given an initialisation.

- template<Descriptor descr, typename OutputType , typename InputType >
RC knn (Vector< OutputType > &u, const Matrix< InputType > &A, const size_t source, const size_t k, Vector< bool > &buf1)

  Given a graph and a source vertex, indicates which vertices are contained within k hops.

- template<Descriptor descr = descriptors::no_operation, typename IOType = double, class Operator = operators::square_diff< IOType, IOType, IOType >>
RC kpp_initialisation (Matrix< IOType > &K, const Matrix< IOType > &X, const Operator &dist_op=Operator())

  a simple implementation of the k++ initialisation algorithm for kmeans

- template<typename IOType >
RC label (Vector< IOType > &out, const Vector< IOType > &y, const Matrix< IOType > &W, const size_t n, const size_t l, const size_t maxIterations=1000)

  The label propagation algorithm.

- template<Descriptor descr, class Ring , typename IOType , typename InputType >
RC mpv (Vector< IOType > &u, const Matrix< InputType > &A, const size_t k, const Vector< IOType > &v, Vector< IOType > &temp, const Ring &ring)

  The matrix powers kernel.

- template<Descriptor descr = descriptors::no_operation, class Ring , typename InputType , typename OutputType , Backend backend, typename Coords >
RC norm2 (OutputType &x, const Vector< InputType, backend, Coords > &y, const Ring &ring=Ring(), const typename std::enable_if< std::is_floating_point< OutputType >::value, void >::type *const =nullptr)

Provides a generic implementation of the 2-norm computation.

- template<Descriptor descr = descriptors::no_operation, typename IOType , typename NonzeroT >
  RC simple_pagerank (Vector< IOType > &pr, const Matrix< NonzeroT > &L, Vector< IOType > &pr_next, Vector< IOType > &pr_nextnext, Vector< IOType > &row_sum, const IOType alpha=0.85, const IOType conv=0.0000001, const size_t max=1000, size_t *const iterations=nullptr, double *const quality=nullptr)

  The canonical PageRank algorithm.
- template<Descriptor descr = descriptors::no_operation, typename IOType , typename WeightType , typename BiasType , typename ThresholdType = IOType, class MinMonoid = Monoid< grb::operators::min< IOType >, grb::identities::infinity >, class ReluMonoid = Monoid< grb::operators::relu< IOType >, grb::identities::negative_infinity >, class Ring = Semiring< grb::operators::add< IOType >, grb::operators::mul< IOType >, grb::identities::zero, grb::identities::one >>
  grb::RC sparse_nn_single_inference (grb::Vector< IOType > &out, const grb::Vector< IOType > &in, const std::vector< grb::Matrix< WeightType > > &layers, const std::vector< BiasType > &biases, const ThresholdType threshold, grb::Vector< IOType > &temp, const ReluMonoid &relu=ReluMonoid(), const Min↩Monoid &min=MinMonoid(), const Ring &ring=Ring())

  Performs an inference step of a single data element through a Sparse Neural Network defined by num_layers sparse weight matrices and num_layers biases.
- template<Descriptor descr = descriptors::no_operation, typename IOType , typename WeightType , typename BiasType , class Relu↩Monoid = Monoid< grb::operators::relu< IOType >, grb::identities::negative_infinity >, class Ring = Semiring< grb::operators::add< IOType >, grb::operators::mul< IOType >, grb::identities::zero, grb::identities::one >>
  grb::RC sparse_nn_single_inference (grb::Vector< IOType > &out, const grb::Vector< IOType > &in, const std::vector< grb::Matrix< WeightType > > &layers, const std::vector< BiasType > &biases, grb::Vector< IOType > &temp, const ReluMonoid &relu=ReluMonoid(), const Ring &ring=Ring())

  Performs an inference step of a single data element through a Sparse Neural Network defined by num_layers sparse weight matrices and num_layers biases.
- template<bool normalize = false, typename IOType >
  RC spy (grb::Matrix< IOType > &out, const grb::Matrix< bool > &in)

  Specialisation for boolean input matrices in.
- template<bool normalize = false, typename IOType , typename InputType >
  RC spy (grb::Matrix< IOType > &out, const grb::Matrix< InputType > &in)

  Given an input matrix and a smaller output matrix, map nonzeroes from the input matrix into the smaller one and count the number of nonzeroes that are mapped from the bigger matrix into the smaller.
- template<bool normalize = false, typename IOType >
  RC spy (grb::Matrix< IOType > &out, const grb::Matrix< void > &in)

  Specialisation for void input matrices in.

8.2.1 Detailed Description

The namespace for ALP/GraphBLAS algorithms.

8.2.2 Function Documentation

8.2.2.1 bicgstab()

```
RC grb::algorithms::bicgstab (
            grb::Vector< IOType > & x,
            const grb::Matrix< NonzeroType > & A,
            const grb::Vector< InputType > & b,
            const size_t max_iterations,
            ResidualType tol,
```

```
size_t & iterations,
ResidualType & residual,
Vector< InputType > & r,
Vector< InputType > & rhat,
Vector< InputType > & p,
Vector< InputType > & v,
Vector< InputType > & s,
Vector< InputType > & t,
const Semiring & semiring = Semiring(),
const Minus & minus = Minus(),
const Divide & divide = Divide() )
```

Solves a linear system $b = Ax$ with x unknown by using the bi-conjugate gradient (bi-CG) stabilised method; i.e., BiCGstab.

Template Parameters

descr	Any descriptor to use for the computation (optional).
IOType	The solution vector element type.
NonzeroType	The system matrix entry type.
InputType	The element type of the right-hand side vector.
ResidualType	The type of the residuals used during computation.
Semiring	The semiring under which to perform the BiCGstab
Minus	The inverse operator of the additive operator of *Semiring*.
Divide	The inverse of the multiplicative operator of *Semiring*.

By default, these will be the regular add, mul, subtract, and divide over the types *IOType*, *NonzeroType*, *InputType*, and/or *ResidualType*, as appropriate.

Does not perform any preconditioning.

Parameters

in,out	*x*	On input↩: an initial guess to the solution $Ax = b$. On output↩: if grb::SUCCESS is returned, the solution to $Ax = b$ within the given tolerance *tol*. Otherwise, the last computed approximation to the solution is returned.
in	*A*	The square non-singular system matrix A.
in	*b*	The right-hand side vector b.

If the size of A is $n \times n$, then the sizes of *x* and *b* must be n also. The vector *x* must have capacity n.

Mandatory inputs to the BiCGstab algorithm:

Parameters

in	*max_iterations*	The maximum number of iterations this algorithm may perform.
in	*tol*	The relative tolerance which determines when an an approximated solution x becomes acceptable. Must be positive and non-zero.

Additional outputs of this algorithm:

Parameters

out	*iterations*	When grb::SUCCESS is returned, the number of iterations that were required to obtain an acceptable approximate solution.
out	*residual*	When grb::SUCCESS is returned, the square of the 2-norm of the residual; i.e., (r,r), where $r = b-Ax$.

To operate, this algorithm requires a workspace consisting of six vectors of length and capacity n. If vectors with less capacity are passed as arguments, grb::ILLEGAL will be returned.

Parameters

in	*r,rhat,p,v,s,t*	Workspace vectors required for Bi↩ CGstab.

The BiCGstab operates over a field defined by the following algebraic structures:

Parameters

`in`	*semiring*	Defines the domains as well as the additive and the multicative monoid.
`in`	*minus*	The inverse of the additive operator.
`in`	*divide*	The inverse of the multiplicative operator.

Note

When compiling with the `_DEBUG` macro defined, the print-out statements require `sqrt` as an additional algebraic concept. This concept presently lives "outside" of ALP.

Valid descriptors to this algorithm are:

1. descriptors::no_casting
2. descriptors::transpose

Returns

grb::SUCCESS If an acceptable solution is returned.

grb::FAILED If the algorithm failed to find an acceptable solution and returns an approximate one with the given *residual*.

grb::MISMATCH If two or more of the input arguments have incompatible sizes.

grb::MISMATCH If one or more of the workspace vectors has an incompatible size.

grb::ILLEGAL If *tol* is zero or negative.

grb::ILLEGAL If *x* has capacity less than n.

grb::ILLEGAL If one or more of the workspace vectors has a capacity less than n.

grb::PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/Graph↩ BLAS is undefined.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.2 conjugate_gradient()

```
grb::RC grb::algorithms::conjugate_gradient (
            grb::Vector< IOType > & x,
            const grb::Matrix< NonzeroType > & A,
            const grb::Vector< InputType > & b,
            const size_t max_iterations,
            ResidualType tol,
            size_t & iterations,
            ResidualType & residual,
            grb::Vector< IOType > & r,
            grb::Vector< IOType > & u,
            grb::Vector< IOType > & temp,
            const Ring & ring = Ring(),
            const Minus & minus = Minus(),
            const Divide & divide = Divide() )
```

Solves a linear system $b = Ax$ with x unknown by the Conjugate Gradients (CG) method on general fields.

Does not perform any preconditioning.

Template Parameters

descr	The user descriptor
IOType	The input/output vector nonzero type
ResidualType	The type of the residual
NonzeroType	The matrix nonzero type
InputType	The right-hand side vector nonzero type
Ring	The semiring under which to perform CG
Minus	The minus operator corresponding to the inverse of the additive operator of the given *Ring*.
Divide	The division operator corresponding to the inverse of the multiplicative operator of the given *Ring*.

Valid descriptors to this algorithm are:

1. descriptors::no_casting
2. descriptors::transpose

By default, i.e., if none of *ring*, *minus*, or *divide* (nor their types) are explicitly provided by the user, the natural field on double data types will be assumed.

Note

An abstraction of a field that encapsulates *Ring*, *Minus*, and *Divide* may be more appropriate. This will also naturally ensure that demands on domain types are met.

Parameters

in,out	*x*	On input↩: an initial guess to the solution. On output↩: the last computed approximation.
in	*A*	The (square) positive semi-definite system matrix.
in	*b*	The known right-hand side in $Ax = b$. Must be structurally dense.

If *A* is $n \times n$, then *x* and *b* must have matching length n. The vector *x* furthermore must have a capacity of n.

CG algorithm inputs:

Parameters

in	*max_iterations*	The maximum number of CG iterations.
in	*tol*	The requested relative tolerance.

Additional outputs (besides *x*):

Parameters

out	*iterations*	The number of iterations the algorithm has started.
out	*residual*	The residual corresponding to output *x*.

The CG algorithm requires three workspace buffers with capacity n:

Parameters

in,out	*r*	A temporary vector of the same size as *x*.

Parameters

in,out	*u*	A temporary vector of the same size as *x*.
in,out	*temp*	A temporary vector of the same size as *x*.

Finally, the algebraic structures over which the CG is executed are given:

Parameters

in	*ring*	The semiring under which to perform the CG.
in	*minus*	The inverse of the additive operator of *ring*.
in	*divide*	The inverse of the multiplicative operator of *ring*.

This algorithm may return one of the following error codes:

Returns

grb::SUCCESS When the algorithm has converged to a solution within the given *max_iterations* and *tol*.

grb::FAILED When the algorithm did not converge within the given *max_iterations*.

grb::ILLEGAL When *A* is not square.

grb::MISMATCH When *x* or *b* does not match the size of *A*.

grb::ILLEGAL When *x* does not have capacity n.

grb::ILLEGAL When at least one of the workspace vectors does not have capacity n.

grb::ILLEGAL If *tol* is not strictly positive.

grb::PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/Graph↩ BLAS is undefined.

On output, the contents of the workspace *r*, *u*, and *temp* are always undefined. For non-grb::SUCCESS error codes, additional containers or states may be left undefined:

1. when grb::PANIC is returned, the entire program state, including the contents of all containers, become undefined;
2. when grb::ILLEGAL or grb::MISMATCH are returned and *iterations* equals zero, then all outputs are left unmodified compared to their contents at function entry;
3. when grb::ILLEGAL or grb::MISMATCH are returned and *iterations* is nonzero, then the contents of *x* are undefined.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.3 cosine_similarity()

```
RC grb::algorithms::cosine_similarity (
            OutputType & similarity,
            const Vector< InputType1 > & x,
            const Vector< InputType2 > & y,
            const Ring & ring = Ring(),
            const Division & div = Division() )
```

Computes the cosine similarity.

Given two vectors x, y of equal size n, this function computes $\alpha = \frac{(x,y)}{||x||_2 \cdot ||y||_2}$.

The 2-norms and inner products are computed according to the given semi- ring. However, the norms make use of the standard `sqrt` and so the algorithm assumes a regular field is used. Effectively, hence, the semiring controls the precision / data types under which the computation is performed.

Template Parameters

descr	The descriptor under which to perform the computation.
OutputType	The type of the output element (scalar).
InputType1	The type of the first vector.
InputType2	The type of the second vector.
Ring	The semiring used.
Division	Which binary operator correspond to division corresponding to the given *Ring*.

Parameters

out	*similarity*	Where to fold the result into.
in	*x*	The non-zero left-hand input vector.
in	*y*	The non-zero right-hand input vector.
in	*ring*	The semiring to compute over.
in	*div*	The division operator corresponding to *ring*.

Note

The vectors *x* and/or *y* may be sparse or dense.

The argument *div* is optional. It will map to grb::operators::divide by default.

Returns

grb::SUCCESS If the computation was successful.

grb::MISMATCH If the vector sizes do not match. The output *similarity* is untouched – the call to this algorithm will have no other effects than returning grb::MISMATCH.

grb::ILLEGAL In case *x* is all zero, and/or when *y* is all zero. The output *similarity* is undefined.

grb::PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/Graph↩ BLAS is undefined.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.4 kcore_decomposition()

```
RC grb::algorithms::kcore_decomposition (
            const Matrix< NZType > & A,
            Vector< IOType > & core,
            Vector< IOType > & distances,
            Vector< IOType > & temp,
            Vector< IOType > & update,
            Vector< bool > & status,
            IOType & k )
```

The k-core decomposition algorithm.

Note

This algorithm is smoke-tested using a ground-truth output coreness vector corresponding to the EPA matrix. However, the ground truth was generated using an earlier version of this algorithm, run using an earlier version of ALP/GraphBLAS. This solution was manually verified against an external algorithm. A better testing methodology compares against a ground truth generated by such an external baseline– see GitHub issue #160, to which contributions would be warmly received.

Divides the input matrix into subgraphs with a coreness level. The coreness level k is defined as the largest subgraph in which each node has at least k neighbors in the subgraph.

Template Parameters

IOType	The value type of the k-core vectors, usually an integer type.
NZType	The type of the nonzero elements in the matrix.

Parameters

in	*A*	Matrix representing a graph with nonzero value at (i, j) an edge between node i and j.
out	*core*	Empty vector of size and capacity n. On output, if grb::SUCCESS is returned, stores the coreness level for each node.
out	*k*	The number of coreness lever that was found in the graph.

To operate, this algorithm requires a workspace of four vectors. The size *and* capacities of these must equal n. The contents on input are ignored, and the contents on output are undefined. The work space consists of the buffer vectors *distances*, *temp*, *update*, and *status*.

Parameters

in,out	*distances*	Distance buffer

Parameters

`in,out`	*temp*	First node update buffer
`in,out`	*update*	Second node update buffer
`in,out`	*status*	Finished/unfinished buffer

Returns

grb::SUCCESS If the coreness for all nodes are found.

grb::ILLEGAL If *A* is not square. All outputs are left untouched.

grb::MISMATCH If the dimensions of *core* or any of the buffer vectors does not match *A*. All outputs are left untouched.

grb::ILLEGAL If the capacity of one or more of *core* and the buffer vectors is less than n.

grb::PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/Graph↩ BLAS is undefined.

If any non grb::SUCCESS error code is returned, then the contents of *core* are undefined, while *k* will be untouched by the algorithm.

Note

For undirected, unweighted graphs, use pattern matrix for *A*; i.e., use *NZtype* `void`

For unweighted graphs, IOType should be a form of unsigned integer. The value of any IOType element will be no more than the maximum degree found in the graph *A*.

Template Parameters

criticalSection	The original MR had an eWiseLambda-based implementation that contains a critical section. This may or may not

Note

In some non-exhaustive experiments, setting *criticalSection* to `false` leads to better performance on shared-memory parallel systems (using grb::reference_omp).

Warning

Setting *criticalSection* to `true` is not supported for the distributed-memory backends grb::BSP1D and grb::hybrid; see the corresponding code comment in the below algorithm for details.

For the above considerations, the default for *criticalSection* is presently set to `false`.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For additional performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

This algorithm is modelled after Li et al., "The K-Core Decomposition Algorithm Under the Framework of Graph↩ BLAS", 2021 IEEE High Performance Extreme Computing Conference (HPEC), doi: 10.1109/HPEC49654.2021.↩ 9622845.

8.2.2.5 kmeans_iteration()

```
RC grb::algorithms::kmeans_iteration (
            Matrix< IOType > & K,
            Vector< std::pair< size_t, IOType > > & clusters_and_distances,
            const Matrix< IOType > & X,
            const size_t max_iter = 1000,
            const Operator & dist_op = Operator() )
```

The kmeans iteration given an initialisation.

Parameters

in,out	*K*	k by m matrix containing the current k means as row vectors
in	*clusters_and_distances*	Vector containing the class and distance to centroid for each point

Parameters

in	*X*	m by n matrix containing the n points to be classified as column vectors
in	*max_iter*	Maximum number of iterations
in	*dist_op*	Coordinatewise distance operator, squared difference by default

8.2.2.6 knn()

```
RC grb::algorithms::knn (
            Vector< OutputType > & u,
            const Matrix< InputType > & A,
            const size_t source,
            const size_t k,
            Vector< bool > & buf1 )
```

Given a graph and a source vertex, indicates which vertices are contained within *k* hops.

This implementation is based on the matrix powers kernel over a Boolean semiring.

Parameters

out	*u*	The distance-k neighbourhood. Any prior contents will be ignored.
in	*A*	The input graph in (square) matrix form
in	*source*	The source vertex index.
in	*k*	The neighbourhood distance, or the maximum number of hops in a breadth-first search.

This algorithm requires the following workspace:

Parameters

in,out	*buf1*	A buffer vector. Must match the size of *A*.

For $n \times n$ matrices *A*, the capacity of *u*, *buf1*, and *buf2* must equal n.

Returns

grb::SUCCESS When the computation completes successfully.

grb::MISMATCH When the dimensions of *u* do not match that of *A*.

grb::MISMATCH If *source* is not in range of *A*.

grb::ILLEGAL If one or more of *u*, *buf1*, or *buf2* has insufficient capacity.

grb::PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/Graph↩ BLAS is undefined.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.7 kpp_initialisation()

```
RC grb::algorithms::kpp_initialisation (
            Matrix< IOType > & K,
            const Matrix< IOType > & X,
            const Operator & dist_op = Operator() )
```

a simple implementation of the k++ initialisation algorithm for kmeans

Parameters

in,out	*K*	k by m matrix containing the current k means as row vectors
in	*X*	m by n matrix containing the n points to be classified as column vectors

Parameters

in	*dist_op*	Coordinatewise distance operator, squared difference by default

8.2.2.8 label()

```
RC grb::algorithms::label (
            Vector< IOType > & out,
            const Vector< IOType > & y,
            const Matrix< IOType > & W,
            const size_t n,
            const size_t l,
            const size_t maxIterations = 1000 )
```

The label propagation algorithm.

Template Parameters

IOType	The value type of the label vector. This will determine the precision of all computations this algorithm performs.

Parameters

out	*out*	The resulting labelled vector representing the n vertices.

Parameters

in	*y*	Vector holding the initial labels from a total set of *n* vertices. The initial labels are assumed to correspond to the vertices corresponding to the first *l* entries of this vector. The labels must be either 0 ar 1.

Parameters

in	*W*	Sparse symmetric matrix of size *n* by *n*, holding the weights between the n vertices. The weights must be positive (larger than 0). The matrix may be defective while the corresponding graph may not be connected.

Parameters

in	*n*	The total number of vertices. If zero, and all of *out*, *y*, and *W* are empty, calling this function is equivalent to a no-op.
in	*l*	The number of vertices with an initial label. Must be larger than zero.
in	*maxIterations*	The maximum number of iterations this algorithm may execute. Optional. Default value↩ : 1000.

Note

If the underlying graph is not connected then some components may be rendered immutable by this algorithm.

Returns

grb::ILLEGAL If one of the arguments passed to this function is illegal. The output will be left unmodified.

grb::ILLEGAL The vector *out* did not have full capacity available. The output will be left unmodified.

grb::ILLEGAL If *n* was nonzero but *l* was zero.

grb::SUCCESS If the computation converged within *max* iterations, or if *n* was zero.

grb::FAILED If the method did not converge within *max* iterations. The output will contain the latest iterand.

grb::PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/Graph↩ BLAS is undefined.

[1] Kamvar, Haveliwala, Manning, Golub; 'Extrapolation methods for accelerating the PageRank computation', ACM Press, 2003.

8.2.2.9 mpv()

```
RC grb::algorithms::mpv (
            Vector< IOType > & u,
            const Matrix< InputType > & A,
            const size_t k,
            const Vector< IOType > & v,
            Vector< IOType > & temp,
            const Ring & ring )
```

The matrix powers kernel.

Calculates $y = A^k x$ for some integer $k \geq 0$ using the given semiring.

Template Parameters

descr	The descriptor used to perform this operation.
Ring	The semiring used.
IOType	The output vector type.
InputType	The nonzero type of matrix elements.
implementation	Which implementation to use.

Parameters

out	*u*	The output vector. Contents shall be overwritten. The supplied vector must match the row dimension size of *A*.
in	*A*	The square input matrix A. The supplied matrix must match the dimensions of *u* and *v*.
in	*k*	How many matrix–vector multiplications are requested.

Parameters

in	*v*	The input vector v. The supplied vector must match the column dimension size of *A*. It may not be the same vector as *u*.
in	*ring*	The semiring to be used. This defines the additive and multiplicative monoids to be used.

This algorithm requires workspace:

Parameters

in,out	*temp*	A workspace buffer of matching size to the row dimension of *A*. May be the same vector as *v*, though note that the contents of *temp* on output are undefined.

This algorithm assumes that *u* and *temp* have full capacity. If this assumption does not hold, then a two-stage mpv must be employed instead.

Returns

grb::SUCCESS If the computation completed successfully.

grb::ILLEGAL If *A* is not square.

grb::MISMATCH If one or more of *u*, *v*, or *temp* has an incompatible size with *A*.

grb::ILLEGAL If one or more of *u* or *temp* does not have a full capacity.

grb::PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/Graph↩ BLAS is undefined.

grb::OVERLAP If one or more of *v* or *temp* is the same vector as *u*.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.10 norm2()

```
RC grb::algorithms::norm2 (
            OutputType & x,
            const Vector< InputType, backend, Coords > & y,
            const Ring & ring = Ring(),
            const typename std::enable_if< std::is_floating_point< OutputType >::value, void
>::type * const  = nullptr )
```

Provides a generic implementation of the 2-norm computation.

Proceeds by computing a dot-product on itself and then taking the square root of the result.

This function is only available when the output type is floating point.

For return codes, exception behaviour, performance semantics, template and non-template arguments,

See also

grb::dot.

Parameters

out	*x*	The 2-norm of *y*. The input value of *x* will be ignored.
in	*y*	The vector to compute the norm of.
in	*ring*	The Semiring under which the 2-norm is to be computed.

Warning

This function computes *x* out-of-place. This is contrary to standard ALP/GraphBLAS functions that are always in-place.

A *ring* is not sufficient for computing a two-norm. This implementation assumes the standard `sqrt` function must be applied on the result of a dot-product of *y* with itself under the supplied semiring.

8.2.2.11 simple_pagerank()

```
RC grb::algorithms::simple_pagerank (
            Vector< IOType > & pr,
            const Matrix< NonzeroT > & L,
            Vector< IOType > & pr_next,
            Vector< IOType > & pr_nextnext,
            Vector< IOType > & row_sum,
            const IOType alpha = 0.85,
            const IOType conv = 0.0000001,
            const size_t max = 1000,
            size_t *const iterations = nullptr,
            double *const quality = nullptr )
```

The canonical PageRank algorithm.

Template Parameters

descr	The descriptor under which to perform the computation.
IOType	The value type of the pagerank vector. This will determine the precision of all computations this algorithm performs.
NonzeroT	The type of the elements of the nonzero matrix.

Parameters

Parameters

in,out	*pr*	Vector of size and capacity n, where n is the vertex size of the input graph *L*. On input, the contents of this vector will be taken as the initial guess to the final result, but only if the vector is dense; if it is, all entries of the initial guess must be nonzero, while it is not, this algorithm will make an initial guess. On output, if grb::SUCCESS is returned, the Page↩ Rank

Parameters

in	*L*	The input graph as a square link matrix of size n.

To operate, this algorithm requires a workspace of three vectors. The size *and* capacities of these must equal n. The contents on input are ignored, and the contents on output are undefined.

This algorithm does not explicitly materialise the Google matrix $G = \alpha L + (1 - \alpha)ee^T$ over which the power iterations are exectuted.

Parameters

in,out	*pr_next*	Buffer for the Page↩ Rank algorithm.
in,out	*pr_nextnext*	Buffer for the Page↩ Rank algorithm.
in,out	*row_sum*	Buffer for the Page↩ Rank algorithm.

The PageRank algorithm holds the following *optional* parameters:

Parameters

in	*alpha*	The scaling factor. The default value is 0.↩ 85. This value must be smaller than 1, and larger than 0.

Parameters

in	*conv*	If the difference between two successive iterations, in terms of its one-norm, is less than this value, then the Page↩ Rank vector is considered converged and this algorithm exits successfully. The default value is 10^{-8}. If this value is set to zero, then the algorithm will continue until *max* iterations are reached. May not be

Parameters

in	*max*	The maximum number of power iterations. The default value is 1000. This value must be larger than 0.

The PageRank algorithm reports the following *optional* output:

Parameters

out	*iterations*	If not nullptr, the number of iterations the call to this algorithm took will be written to the location pointed to.
out	*quality*	If not nullptr, the last computed residual will be written to the location pointed to.

Returns

grb::SUCCESS If the computation converged within *max* iterations.

grb::ILLEGAL If *L* is not square. All outputs are left untouched.

grb::MISMATCH If the dimensions of *pr* and *L* do not match. All outputs are left untouched.

grb::ILLEGAL If an invalid value for *alpha*, *conv*, or *max* was given. All outputs are left untouched.

grb::ILLEGAL If the capacity of one or more of *pr*, *pr_next*, *pr_nextnext*, or *row_sum* is less than n. All outputs are left untouched.

grb::FAILED If the PageRank method did not converge to within the given tolerance *conv* within the given maximum number of iterations *max*. The output *pr* is the last computed approximation, while *iterations* and *quality* are likewise set to *max* and the last computed residual, respectively.

grb::PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/Graph↩ BLAS is undefined.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.12 sparse_nn_single_inference() [1/2]

```
grb::RC grb::algorithms::sparse_nn_single_inference (
            grb::Vector< IOType > & out,
            const grb::Vector< IOType > & in,
            const std::vector< grb::Matrix< WeightType > > & layers,
            const std::vector< BiasType > & biases,
            const ThresholdType threshold,
            grb::Vector< IOType > & temp,
            const ReluMonoid & relu = ReluMonoid(),
            const MinMonoid & min = MinMonoid(),
            const Ring & ring = Ring() )
```

Performs an inference step of a single data element through a Sparse Neural Network defined by *num_layers* sparse weight matrices and *num_layers* biases.

The initial single data element may be sparse also, such as common in Graph Neural Networks (GNNs).

Inference here is a repeated sequence of application of a sparse linear layer, addition of a bias factor, and the application of a ReLU.

We employ a linear algebraic formulation where the ReLU and the bias application are jointly applied via a max-operator.

This formalism follows closely the linear algebraic approach to the related IEEE/MIT GraphChallenge problem, such as for example described in

Combinatorial Tiling for Sparse Neural Networks F. Pawlowski, R. H. Bisseling, B. Uçar and A. N. Yzelman 2020 IEEE High Performance Extreme Computing (HPEC) Conference

Parameters

out	*out*	The result of inference through the neural network.
in	*in*	The input vector, may be sparse or dense.
in	*layers*	A collection of linear layers. Each layer is assumed to be square and of the equal size to one another.

This implies that all *layers* are $n \times n$. The vectors *in* and *out* hence must be of length n.

Commonly, as an input propagates through a network, the features become increasingly dense. Hence *out* is assumed to have full capacity in order to potentially store a fully dense activation vector.

Inference proceeds under a set of biases, one for each layer. Activation vectors are added a constant bias value prior to applying the given *relu* function. After application, the resulting vector is furthermore tresholded. The treshold is assumed constant over all layers.

Parameters

in	*biases*	An array of *num*↩ _↩ *layers* bias factors.

Parameters

in	*threshold*	The value used for thresholding.

Inference is done using a single buffer that is alternated with *out:*

Parameters

in,out	*temp*	A buffer of size and capacity n.

Finally, optional arguments define the algebraic structures under which inference proceeds:

Parameters

in	*relu*	The non-linear ReLU function to apply element-wise.
in	*min*	Operator used for thresholding. Maximum feature value is hard-coded to 32, as per the Graph↩ Challenge.

Parameters

in	*ring*	The semiring under which to perform the inference.

The default algebraic structures are standard *relu* (i.e., max), *min* for tresholding, and the real (semi-) *ring*.

Valid descriptors for this algorithm are:

1. descriptor::no_casting

Note

This algorithm applies the propagation through layers in-place. To facilitate this, only square layers are allowed. Non-square layers would require the use of different vectors at every layer.

Thresholding here means that feature maps as propagated through the neural network are capped at some maximum value, *threshold*.

Returns

grb::SUCCESS If the inference was successful

grb::ILLEGAL If the size of *layers* does not match that of *baises*.

grb::MISMATCH If at least one pair of dimensions between *layers*, *in*, *out*, and *temp* do not match.

grb::ILLEGAL If at least one layer was not square.

grb::ILLEGAL If the capacities of one or more of *out* and *temp* were not full.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.13 sparse_nn_single_inference() [2/2]

```
grb::RC grb::algorithms::sparse_nn_single_inference (
            grb::Vector< IOType > & out,
            const grb::Vector< IOType > & in,
            const std::vector< grb::Matrix< WeightType > > & layers,
            const std::vector< BiasType > & biases,
            grb::Vector< IOType > & temp,
            const ReluMonoid & relu = ReluMonoid(),
            const Ring & ring = Ring() )
```

Performs an inference step of a single data element through a Sparse Neural Network defined by *num_layers* sparse weight matrices and *num_layers* biases.

The initial single data element may be sparse also, such as common in Graph Neural Networks (GNNs).

Inference here is a repeated sequence of application of a sparse linear layer, addition of a bias factor, and the application of a ReLU.

We employ a linear algebraic formulation where the ReLU and the bias application are jointly applied via a max-operator.

This formalism follows closely the linear algebraic approach to the related IEEE/MIT GraphChallenge problem, such as, for example, described in

Combinatorial Tiling for Sparse Neural Networks F. Pawlowski, R. H. Bisseling, B. Uçar and A. N. Yzelman 2020 IEEE High Performance Extreme Computing (HPEC) Conference

Parameters

out	*out*	The result of inference through the neural network.
in	*in*	The input vector, may be sparse or dense.

Parameters

in	*layers*	A collection of linear layers. Each layer is assumed to be square and of the equal size to one another.

This implies that all *layers* are $n \times n$. The vectors *in* and *out* hence must be of length n.

Commonly, as an input propagates through a network, the features become increasingly dense. Hence *out* is assumed to have full capacity in order to potentially store a fully dense activation vector.

Inference proceeds under a set of biases, one for each layer. Activation vectors are added a constant bias value prior to applying the given *relu* function. This function does not perform tresholding.

Parameters

in	*biases*	An array of *num*↩ _↩ *layers* bias factors.

Inference is done using a single buffer that is alternated with *out:*

Parameters

in,out	*temp*	A buffer of size and capacity n.

Finally, optional arguments define the algebraic structures under which inference proceeds:

Parameters

in	*relu*	The non-linear ReLU function to apply element-wise.
in	*ring*	The semiring under which to perform the inference.

The default algebraic structures are standard *relu* (i.e., max), *min* for tresholding, and the real (semi-) *ring*.

Valid descriptors for this algorithm are:

1. descriptor::no_casting

Note

This algorithm applies the propagation through layers in-place. To facilitate this, only square layers are allowed. Non-square layers would require the use of different vectors at every layer.

Returns

grb::SUCCESS If the inference was successful

grb::ILLEGAL If the size of *layers* does not match that of *baises*.

grb::MISMATCH If at least one pair of dimensions between *layers*, *in*, *out*, and *temp* do not match.

grb::ILLEGAL If at least one layer was not square.

grb::ILLEGAL If the capacities of one or more of *out* and *temp* were not full.

Performance semantics

1. This function does not allocate nor free dynamic memory, nor shall it make any system calls.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.14 spy() [1/3]

```
RC grb::algorithms::spy (
            grb::Matrix< IOType > & out,
            const grb::Matrix< bool > & in )
```

Specialisation for boolean input matrices *in*.

See grb::algorithms::spy.

8.2.2.15 spy() [2/3]

```
RC grb::algorithms::spy (
            grb::Matrix< IOType > & out,
            const grb::Matrix< InputType > & in )
```

Given an input matrix and a smaller output matrix, map nonzeroes from the input matrix into the smaller one and count the number of nonzeroes that are mapped from the bigger matrix into the smaller.

Template Parameters

normalize	If set to true, will not compute a number of mapped nonzeroes, but its inverse instead (one divided by the count). The

Parameters

`out`	*out*	The smaller output matrix.
`in`	*in*	The larger input matrix.

Returns

SUCCESS If the computation completes successfully.

ILLEGAL If *out* has a number of rows or columns larger than that of *in*.

Warning

Explicit zeroes (that when cast from *InputType* to *bool* read `false`) *will* be counted as a nonzero by this algorithm.

Note

To not count explicit zeroes, pre-process the input matrix *in*, for example, as follows: `grb::set( tmp, in, true );`, with *tmp* a Boolean or pattern matrix of the same size as *in*.

Performance semantics

Warning

This algorithm does NOT request workspace buffers since due to the use of level-3 primitives it will have to allocate anyway– as such, this algorithm does not have clear performance semantics and should be used with care.

For performance semantics regarding work, inter-process data movement, intra-process data movement, synchronisations, and memory use, please see the specification of the ALP primitives this function relies on. These performance semantics, with the exception of getters such as grb::nnz, are specific to the backend selected during compilation.

8.2.2.16 spy() [3/3]

```
RC grb::algorithms::spy (
            grb::Matrix< IOType > & out,
            const grb::Matrix< void > & in )
```

Specialisation for void input matrices *in*.

See grb::algorithms::spy.

8.3 grb::algorithms::pregel Namespace Reference

The namespace for ALP/Pregel algorithms.

Classes

- struct ConnectedComponents

  A vertex-centric Connected Components algorithm.
- struct PageRank

  A Pregel-style PageRank-like algorithm.

8.3.1 Detailed Description

The namespace for ALP/Pregel algorithms.

8.4 grb::config Namespace Reference

Compile-time configuration constants as well as implementation details that are derived from such settings.

Classes

- class ANALYTIC_MODEL

  Configuration parameters relating to the analytic model employed by the nonblocking backend.
- class BENCHMARKING

  Benchmarking default configuration parameters.
- class CACHE_LINE_SIZE

  Contains information about the target architecture cache line size.
- class IMPLEMENTATION

  Collects a series of implementation choices corresponding to some given backend.
- class IMPLEMENTATION< BSP1D >

  This class collects configuration parameters that are specific to the grb::BSP1D and grb::hybrid backends.
- class IMPLEMENTATION< nonblocking >

  Implementation-dependent configuration parameters for the nonblocking backend.
- class IMPLEMENTATION< reference >

  This class collects configuration parameters that are specific to the grb::reference backend.
- class IMPLEMENTATION< reference_omp >

  This class collects configuration parameters that are specific to the grb::reference_omp backend.
- class MEMORY

  Memory configuration parameters.
- class PIPELINE

  Configuration parameters relating to the pipeline data structure.
- class PREFETCHING

  Default prefetching settings for reference and reference_omp backends.
- class SIMD_SIZE

  The SIMD size, in bytes.

Typedefs

- typedef unsigned int ColIndexType

  What data type should be used to store column indices.
- typedef size_t NonzeroIndexType

  What data type should be used to refer to an array containing nonzeroes.
- typedef unsigned int RowIndexType

  What data type should be used to store row indices.
- typedef unsigned int VectorIndexType

  What data type should be used to store vector indices.

Enumerations

- enum ALLOC_MODE { ALIGNED , INTERLEAVED }

  The memory allocation modes implemented in the grb::reference and the grb::reference_omp backends.

Functions

- std::string **toString** (const ALLOC_MODE mode)

  Converts instances of grb::config::ALLOC_MODE to a descriptive lower-case string.

8.4.1 Detailed Description

Compile-time configuration constants as well as implementation details that are derived from such settings.

8.5 grb::descriptors Namespace Reference

Collection of standard descriptors.

Functions

- std::string toString (const Descriptor descr)

  Translates a descriptor into a string.

Variables

- static constexpr Descriptor add_identity = 32

  For any call to a matrix computation, the input matrix A is instead interpreted as $A + I$, with I the identity matrix of dimension matching A.
- static constexpr Descriptor dense = 16

  Indicates that all input and output vectors to an ALP/GraphBLAS primitive are structurally dense.
- static constexpr Descriptor explicit_zero = 512

  Computation shall proceed with zeros (according to the current semiring) propagating throughout the requested computation.
- static constexpr Descriptor **invert_mask** = 1

  Inverts the mask prior to applying it.
- static constexpr Descriptor no_casting = 256

  Disallows the standard casting of input parameters to a compatible domain in case they did not match exactly.
- static constexpr Descriptor no_duplicates = 4

  For data ingestion methods, such as grb::buildVector or grb::buildMatrix, this descriptor indicates that the input shall not contain any duplicate entries.
- static constexpr Descriptor **no_operation** = 0

  Indicates no additional pre- or post-processing on any of the GraphBLAS function arguments.
- static constexpr Descriptor **safe_overlap** = 1024

  Indicates overlapping input and output vectors is intentional and safe, due to, for example, the use of masks.
- static constexpr Descriptor structural = 8

  Uses the structure of a mask vector only.
- static constexpr Descriptor structural_complement = structural | invert_mask

  Uses the structural complement of a mask vector.
- static constexpr Descriptor **transpose_left** = 2048

  For operations involving two matrices, transposes the left-hand side input matrix prior to applying it.
- static constexpr Descriptor **transpose_matrix** = 2

  Transposes the input matrix prior to applying it.
- static constexpr Descriptor **transpose_right** = 4096

  For operations involving two matrices, transposes the right-hand side input matrix prior to applying it.
- static constexpr Descriptor use_index = 64

  Instead of using input vector elements, use the index of those elements.

8.5.1 Detailed Description

Collection of standard descriptors.

8.5.2 Function Documentation

8.5.2.1 toString()

```
std::string grb::descriptors::toString (
            const Descriptor descr )
```

Translates a descriptor into a string.

Parameters

in	*descr*	The input descriptor.

Returns

A detailed English description.

8.5.3 Variable Documentation

8.5.3.1 add_identity

```
constexpr Descriptor add_identity = 32  [static], [constexpr]
```

For any call to a matrix computation, the input matrix *A* is instead interpreted as $A + I$, with *I* the identity matrix of dimension matching *A*.

If *A* is not square, padding zero columns or rows will be added to *I* in the largest dimension.

8.5.3.2 dense

```
constexpr Descriptor dense = 16  [static], [constexpr]
```

Indicates that all input and output vectors to an ALP/GraphBLAS primitive are structurally dense.

If a user passes this descriptor but one or more vectors to the call are *not* structurally dense, then ILLEGAL shall be returned.

Warning

All vectors includes any vectors that operate as masks. Thus if the primitive is to operate with structurally sparse masks but with otherwise dense vectors, then the dense descriptor may *not* be defined.

For in-place operations with vector outputs –which are all ALP/GraphBLAS primitives with vector outputs except grb::set and grb::eWiseApply– the output vector is also an input vector. Thus passing this descriptor to such primitive indicates that also the output vector is structurally dense.

For out-of-place operations with vector output(s), passing this descriptor also demands that the output vectors are already dense.

Vectors with explicit zeroes (under the semiring passed to the related primitive) will be computed with explicitly.

The benefits of using this descriptor whenever possible are two-fold: 1) less run-time overhead as code handling sparsity is disabled; 2) smaller binary sizes as code handling structurally sparse vectors is not emitted (unless required elsewhere).

The consistent use of this descriptor is hence strongly encouraged.

8.5.3.3 explicit_zero

```
constexpr Descriptor explicit_zero = 512  [static], [constexpr]
```

Computation shall proceed with zeros (according to the current semiring) propagating throughout the requested computation.

Warning

This may lead to unexpected results if the same output container is interpreted under a different semiring– what is zero for the current semiring may not be zero for another. In other words: the concept of sparsity will no longer generalise to other semirings.

8.5.3.4 no_casting

```
constexpr Descriptor no_casting = 256  [static], [constexpr]
```

Disallows the standard casting of input parameters to a compatible domain in case they did not match exactly.

Setting this descriptor will yield compile-time errors whenever casting would have been necessary to successfully compile the requested graphBLAS operation.

Warning

It is illegal to perform conditional toggling on this descriptor.

Note

With conditional toggling, if `descr` is a descriptor, we mean `if( descr & descriptors::no_casting ) { new_descr = desc - descriptors::no_casting //followed by any use of this new descriptor }` The reason we cannot allow for this type of toggling is because this descriptor makes use of the `static_assert` C++11 function, which is checked regardless of the result of the if-statement. Thus the above code actually always throws compile errors on mismatching domains, no matter the original value in `descr`.

8.5.3.5 no_duplicates

```
constexpr Descriptor no_duplicates = 4  [static], [constexpr]
```

For data ingestion methods, such as grb::buildVector or grb::buildMatrix, this descriptor indicates that the input shall not contain any duplicate entries.

Use of this descriptor will speed up the corresponding function call significantly.

A call to buildMatrix with this descriptor set will pass its arguments to buildMatrixUnique.

Warning

Use of this descriptor while the data to be ingested actually *does* contain duplicates will lead to undefined behaviour.

Currently, the reference implementation only supports ingesting data using this descriptor. Support for duplicate input is not yet implemented everywhere.

8.5.3.6 structural

```
constexpr Descriptor structural = 8  [static], [constexpr]
```

Uses the structure of a mask vector only.

This ignores the actual values of the mask argument. The i-th element of the mask now evaluates true if the mask has *any* value assigned to its i-th index, regardless of how that value evaluates. It evaluates false if there was no value assigned.

See also

structural_complement

8.5.3.7 structural_complement

```
constexpr Descriptor structural_complement = structural | invert_mask  [static], [constexpr]
```

Uses the structural complement of a mask vector.

This is a convenience short-hand for:

```
constexpr Descriptor structural_complement = structural | invert_mask;
```

This ignores the actual values of the mask argument. The i-th element of the mask now evaluates true if the mask has *no* value assigned to its i-th index, and evaluates false otherwise.

8.5.3.8 use_index

```
constexpr Descriptor use_index = 64  [static], [constexpr]
```

Instead of using input vector elements, use the index of those elements.

Indices are cast from their internal data type (`size_t`, e.g.) to the relevant domain of the operator used.

8.6 grb::identities Namespace Reference

Standard identities common to many operators.

Classes

- class infinity

  Standard identity for the minimum operator.
- class logical_false

  Standard identitity for the logical or operator.
- class logical_true

  Standard identity for the logical AND operator.
- class negative_infinity

  Standard identity for the maximum operator.
- class one

  Standard identity for numerical multiplication.
- class zero

  Standard identity for numerical addition.

8.6.1 Detailed Description

Standard identities common to many operators.

The most commonly used identities are

- grb::identities::zero, and
- grb::identities::one.

A stateful identity should expose the same public interface as the identities collected here, which is class which exposes at least one public templated function named *value*, taking no arguments, returning the identity in the domain *D*. This type *D* is the first template parameter of the function *value*. If there are other template parameters, those template parameters are required to have defaults.

See also

operators

Monoid

Semiring

8.7 grb::interfaces Namespace Reference

The namespace for programming APIs that automatically translate to ALP/GraphBLAS.

Namespaces

- namespace config

  Contains configurations for programming models that are simulated on top of ALP/GraphBLAS.

Classes

- class Pregel

  A Pregel run-time instance.
- struct PregelState

  The state of the vertex-center Pregel program that the user may interface with.

8.7.1 Detailed Description

The namespace for programming APIs that automatically translate to ALP/GraphBLAS.

8.8 grb::interfaces::config Namespace Reference

Contains configurations for programming models that are simulated on top of ALP/GraphBLAS.

Enumerations

- enum SparsificationStrategy { NONE = 0 , ALWAYS , WHEN_REDUCED , WHEN_HALVED }

  The set of sparsification strategies supported by the ALP/Pregel interface.

Variables

- constexpr const SparsificationStrategy **out_sparsify** = NONE

  What sparsification strategy should be applied to the outgoing messages.

8.8.1 Detailed Description

Contains configurations for programming models that are simulated on top of ALP/GraphBLAS.

8.9 grb::operators Namespace Reference

This namespace holds various standard operators such as grb::operators::add and grb::operators::mul.

Classes

- class abs_diff

  This operator returns the absolute difference between two numbers.
- class add

  This operator takes the sum of the two input parameters and writes it to the output variable.
- class any_or

  This operator is a generalisation of the logical or.
- class argmax

  The argmax operator on key-value pairs.
- class argmin

  The argmin operator on key-value pairs.
- class divide

  Numerical division of two numbers.
- class divide_reverse

  Reversed division of two numbers.
- class equal

  Operator which returns `true` if its inputs compare equal, and `false` otherwise.
- class equal_first

  Compares `std::pair` inputs taking the first entry in every pair as the comparison key, and returns `true` or `false` accordingly.
- class geq

  This operation returns whether the left operand compares greater-than or equal to the right operand.
- class greater_than

  This operation returns whether the left operand compares greater-than the right operand.
- class left_assign

  This operator discards all right-hand side input and simply copies the left-hand side input to the output variable.
- class left_assign_if

  This operator assigns the left-hand input if the right-hand input evaluates `true`.
- class leq

  This operation returns whether the left operand compares less-than or equal to the right operand.
- class less_than

  This operation returns whether the left operand compares less-than the right operand.
- class logical_and

  The logical and.
- class logical_or

  The logical or.
- class max

  This operator takes the maximum of the two input parameters and writes the result to the output variable.
- class min

  This operator takes the minimum of the two input parameters and writes the result to the output variable.
- class mul

  This operator multiplies the two input parameters and writes the result to the output variable.
- class not_equal

  Operator that returns `false` whenever its inputs compare equal, and `true` otherwise.
- class relu

  This operation is equivalent to grb::operators::min.
- class right_assign

  This operator discards all left-hand side input and simply copies the right-hand side input to the output variable.
- class right_assign_if

  This operator assigns the right-hand input if the left-hand input evaluates `true`.

- class square_diff

  This operation returns the squared difference between two numbers.
- class subtract

  Numerical substraction of two numbers.
- class zip

  The zip operator that operators on keys as a left-hand input and values as a right hand input, producing a key-value `std::pair`.

8.9.1 Detailed Description

This namespace holds various standard operators such as grb::operators::add and grb::operators::mul.

Chapter 9

Class Documentation

9.1 abs_diff< D1, D2, D3, implementation > Class Template Reference

This operator returns the absolute difference between two numbers.

`#include <ops.hpp>`

Inherits Operator< internal::abs_diff< D1, D1, D1, config::default_backend > >.

9.1.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::abs_diff< D1, D2, D3, implementation >

This operator returns the absolute difference between two numbers.

Mathematical notation: $\odot(x, y) \to |x - y|$.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator- and `std::abs` overloads available.

See also

square_diff

The documentation for this class was generated from the following file:

- ops.hpp

9.2 add< D1, D2, D3, implementation > Class Template Reference

This operator takes the sum of the two input parameters and writes it to the output variable.

`#include <ops.hpp>`

Inherits Operator< internal::add< D1, D1, D1, config::default_backend > >.

9.2.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::add< D1, D2, D3, implementation >

This operator takes the sum of the two input parameters and writes it to the output variable.

It exposes the complete interface detailed in grb::operators::internal::Operator. This operator can be passed to any GraphBLAS function or object constructor.

Mathematical notation: $\odot(x, y) \rightarrow x + y$.

Template Parameters

D1	The left-hand side input domain.
D2	The right-hand side input domain.
D3	The output domain.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator+- functions available.

The documentation for this class was generated from the following file:

- ops.hpp

9.3 ANALYTIC_MODEL Class Reference

Configuration parameters relating to the analytic model employed by the nonblocking backend.

`#include <config.hpp>`

Static Public Attributes

- static constexpr const double **L1_CACHE_USAGE_PERCENTAGE** = 0.98

  The L1 cache size is assumed to be a bit smaller than the actual size to take into account any data that may be stored in cache and are not considered by the analytic model, e.g., matrices for the current design.
- static constexpr const size_t MIN_TILE_SIZE = 512

  The minimum tile size that may be automatically selected by the analytic model.

9.3.1 Detailed Description

Configuration parameters relating to the analytic model employed by the nonblocking backend.

9.3.2 Member Data Documentation

9.3.2.1 MIN_TILE_SIZE

```
constexpr const size_t MIN_TILE_SIZE = 512  [static], [constexpr]
```

The minimum tile size that may be automatically selected by the analytic model.

A tile size that is set manually may be smaller than MIN_TILE_SIZE.

The documentation for this class was generated from the following file:

- nonblocking/config.hpp

9.4 any_or< D1, D2, D3, implementation > Class Template Reference

This operator is a generalisation of the logical or.

`#include <ops.hpp>`

Inherits Operator< internal::any_or< D1, D1, D1, config::default_backend > >.

9.4.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::any_or< D1, D2, D3, implementation >

This operator is a generalisation of the logical or.

It assigns to the output any input which evaluates `true`. If there is no such input, it assigns any input that evaluates `false`.

Note

The main difference is that the output is never cast from a Boolean `true` or `false`.

The input domains must be *castable* to `bool`.

The input domains must furthermore be *castable* to the output domain.

The documentation for this class was generated from the following file:

- ops.hpp

9.5 argmax< IType, VType > Class Template Reference

The argmax operator on key-value pairs.

`#include <ops.hpp>`

Inherits Operator< internal::argmax< IType, VType > >.

9.5.1 Detailed Description

template<typename IType, typename VType>
class grb::operators::argmax< IType, VType >

The argmax operator on key-value pairs.

Template Parameters

IType	The key type.
VType	The value type.

This operator is only defined for key-value pairs encapsulated in the STL standard `std::pair`. The return type equals that of the key type.

This operator returns the key corresponding to the key-value pair whose value evaluates greater than the other.

Warning

If both values are equal, any key may be returned.

See also

argmin

equal_first

The documentation for this class was generated from the following file:

- ops.hpp

9.6 argmin< IType, VType > Class Template Reference

The argmin operator on key-value pairs.

`#include <ops.hpp>`

Inherits Operator< internal::argmin< IType, VType > >.

9.6.1 Detailed Description

template<typename IType, typename VType>
class grb::operators::argmin< IType, VType >

The argmin operator on key-value pairs.

Template Parameters

IType	The key type.
VType	The value type.

This operator is only defined for key-value pairs encapsulated in the STL standard `std::pair`. The return type equals that of the key type.

This operator returns the key corresponding to the key-value pair whose value evaluates less than the other.

Warning

If both values are equal, any key may be returned.

See also

argmax

equal_first

The documentation for this class was generated from the following file:

- ops.hpp

9.7 Benchmarker< mode, implementation > Class Template Reference

A class that follows the API of the grb::Launcher, but instead of launching the given ALP program once, it launches it multiple times while benchmarking its execution times.

`#include <benchmark.hpp>`

Public Member Functions

- Benchmarker (const size_t process_id=0, size_t nprocs=1, std::string hostname="localhost", std::string port="0")

  Constructs an instance of the benchmarker class.
- template<typename T , typename U >
  RC exec (void(*alp_program)(const T &, U &), const T &data_in, U &data_out, const size_t inner, const size_t outer, const bool broadcast=false) const

  Benchmarks a given ALP program.
- template<typename U >
  RC exec (void(*alp_program)(const void *, const size_t, U &), const void *data_in, const size_t in_size, U &data_out, const size_t inner, const size_t outer, const bool broadcast=false) const

  Benchmarks a given ALP program.

Static Public Member Functions

- static RC finalize ()

  Releases all ALP resources.

9.7.1 Detailed Description

template<enum EXEC_MODE mode, enum Backend implementation>
class grb::Benchmarker< mode, implementation >

A class that follows the API of the grb::Launcher, but instead of launching the given ALP program once, it launches it multiple times while benchmarking its execution times.

See also

Benchmarking

9.7.2 Constructor & Destructor Documentation

9.7.2.1 Benchmarker()

```
Benchmarker (
            const size_t process_id = 0,
            size_t nprocs = 1,
            std::string hostname = "localhost",
            std::string port = "0" )  [inline]
```

Constructs an instance of the benchmarker class.

Parameters

in	*process_id*	A unique ID for the calling user process.

Parameters

in	*nprocs*	The total number of user processes participating in the benchmark. The given *process_id* must be strictly smaller than this given value.
in	*hostname*	The hostname where one of the user processes participating in the benchmark resides.
in	*port*	A free TCP/IP port at the host corresponding to the given *hostname*.

The *hostname* and *port* arguments are unused if *nprocs* equals one.

All arguments are optional– their defaults are:

- 0 for *process_id*,
- 1 for *nprocs*,
- *localhost* for *hostname*, and
- 0 for *port*.

This constructor may throw the same errors as grb::Launcher.

See also

grb::Launcher

Benchmarking

9.7.3 Member Function Documentation

9.7.3.1 exec() [1/2]

```
RC exec (
            void(*)(const T &, U &) alp_program,
            const T & data_in,
            U & data_out,
            const size_t inner,
            const size_t outer,
            const bool broadcast = false ) const  [inline]
```

Benchmarks a given ALP program.

This variant applies to input data as a user-defined POD struct and output data as a user-defined POD struct.

Template Parameters

T	Input type of the given user program.
U	Output type of the given user program.

Parameters

in	*alp_program*	The ALP program to be benchmarked
in	*data_in*	Input data as a raw data blob

Parameters

out	*data_out*	Output data
in	*inner*	The number of inner repetitions of the benchmark
in	*outer*	The number of outer repetitions of the benchmark
in	*broadcast*	An optional argument that dictates whether the *data_in* argument should be broadcast across all user processes participating in the benchmark, prior to *each* invocation of *alp_program*.

The default value of *broadcast* is `false`.

Returns

grb::SUCCESS The benchmarking has completed successfully.

grb::FAILED An error during benchmarking has occurred. The benchmark attempt could be retried, and an error for the failure is reported to the standard error stream.

grb::PANIC If an unrecoverable error was encountered while starting the benchmark, while benchmarking, or while aggregating the final results.

See also

Benchmarking

9.7.3.2 exec() [2/2]

```
RC exec (
            void(*)(const void *, const size_t, U &) alp_program,
            const void * data_in,
            const size_t in_size,
            U & data_out,
            const size_t inner,
            const size_t outer,
            const bool broadcast = false ) const  [inline]
```

Benchmarks a given ALP program.

This variant applies to input data as a byte blob and output data as a user-defined POD struct.

Template Parameters

U	Output type of the given user program.

Parameters

in	*alp_program*	The user program to be benchmarked
in	*data_in*	Input data as a raw data blob
in	*in_size*	The size, in bytes, of the input data

Parameters

out	*data_out*	Output data
in	*inner*	The number of inner repetitions of the benchmark
in	*outer*	The number of outer repetitions of the benchmark
in	*broadcast*	An optional argument that dictates whether the *data_in* argument should be broadcast across all user processes participating in the benchmark, prior to *each* invocation of *alp_program*.

The default value of *broadcast* is `false`.

Returns

grb::SUCCESS The benchmarking has completed successfully.

grb::ILLEGAL If *in_size* is nonzero but *data_in* compares equal to `nullptr`.

grb::FAILED An error during benchmarking has occurred. The benchmark attempt could be retried, and an error for the failure is reported to the standard error stream.

grb::PANIC If an unrecoverable error was encountered while starting the benchmark, while benchmarking, or while aggregating the final results.

See also

Benchmarking

9.7.3.3 finalize()

```
static RC finalize ( )  [inline], [static]
```

Releases all ALP resources.

Calling this function is equivalent to calling grb::Launcher::finalize.

After a call to this function, no further ALP programs may be benchmarked nor launched– i.e., both the grb::Launcher and grb::Benchmarker functionalities many no longer be used.

A well-behaving program calls this function, or grb::Launcher::finalize, exactly once and just before exiting (or just before the guaranteed last invocation of an ALP program).

Returns

grb::SUCCESS The resources have successfully and permanently been released.

grb::PANIC An unrecoverable error has been encountered and the user program is encouraged to exit as quickly as possible. The state of the ALP library has become undefined and should no longer be used.

The documentation for this class was generated from the following file:

- benchmark.hpp

9.8 BENCHMARKING Class Reference

Benchmarking default configuration parameters.

```
#include <config.hpp>
```

Static Public Member Functions

- static constexpr size_t inner ()
- static constexpr size_t outer ()

9.8.1 Detailed Description

Benchmarking default configuration parameters.

The documentation for this class was generated from the following file:

- base/config.hpp

9.9 CACHE_LINE_SIZE Class Reference

Contains information about the target architecture cache line size.

`#include <config.hpp>`

9.9.1 Detailed Description

Contains information about the target architecture cache line size.

The documentation for this class was generated from the following file:

- base/config.hpp

9.10 collectives< implementation > Class Template Reference

A static class defining various collective operations on scalars.

`#include <collectives.hpp>`

Static Public Member Functions

- template<Descriptor descr = descriptors::no_operation, typename Operator , typename IOType >
  static RC allreduce (IOType &inout, const Operator op=Operator())

  Schedules an allreduce operation of a single object of type IOType per process.
- template<typename IOType >
  static RC broadcast (IOType &inout, const size_t root=0)

  Schedules a broadcast operation of a single object of type IOType per process.
- template<Descriptor descr = descriptors::no_operation, typename IOType >
  static RC broadcast (IOType *inout, const size_t size, const size_t root=0)

  Broadcast on an array of IOType.
- template<Descriptor descr = descriptors::no_operation, typename Operator , typename IOType >
  static RC reduce (IOType &inout, const size_t root=0, const Operator op=Operator())

  Schedules a reduce operation of a single object of type IOType per process.

9.10.1 Detailed Description

template<enum Backend implementation>
class grb::collectives< implementation >

A static class defining various collective operations on scalars.

This class is templated in terms of the backends that are implemented– each implementation provides its own mechanisms to handle collective communications. These are required for users employing grb::eWiseLambda, or for users who perform explicit SPMD programming.

9.10.2 Member Function Documentation

9.10.2.1 allreduce()

```
static RC allreduce (
            IOType & inout,
            const Operator op = Operator() )  [inline], [static]
```

Schedules an allreduce operation of a single object of type IOType per process.

The allreduce shall be complete by the end of the call. This is a collective graphBLAS operation. After the collective call finishes, each user process will locally have available the allreduced value.

Since this is a collective call, there are *P* values *inout* spread over all user processes. Let these values be denoted by x_s, with $s \in \{0, 1, \ldots, P-1\}$, such that x_s equals the argument *inout* on input at the user process with ID *s*. Let $\pi : \{0, 1, \ldots, P-1\} \to \{0, 1, \ldots, P-1\}$ be a bijection, some unknown permutation of the process ID. This permutation is must be fixed for any given combination of GraphBLAS implementation and value *P*. Let the binary operator *op* be denoted by $\odot$.

This function computes $\odot_{i=0}^{P-1} x_{\pi(i)}$ and writes the exact same result to *inout* at each of the *P* user processes.

In summary, this means 1) this operation is coherent across all processes and produces bit-wise equivalent output on all user processes, and 2) the result is reproducible across different runs using the same input and *P*. Yet it does *not* mean that the order of addition is fixed.

Since each user process supplies but one value, there is no difference between a reduce-to-the-left versus a reduce-to-the-right (see grb::reducel and grb::reducer).

Template Parameters

descr	The GraphBLAS descriptor. Default is grb::descriptors::no_operation.
Operator	Which operator to use for reduction.
IOType	The type of the to-be reduced value.

Parameters

in,out	*inout*	On input↩ : the value at the calling process to be reduced. On output↩ : the reduced value.
in	*op*	The associative operator to reduce by.

Note

If *op* is commutative, the implementation free to employ a different allreduce algorithm, as long as it is documented well enough so that its cost can be quantified.

Returns

grb::SUCCESS When the operation succeeds as planned.

grb::PANIC When the communication layer unexpectedly fails. When this error code is returned, the library enters an undefined state.

Valid descriptors:

1. grb::descriptors::no_operation
2. grb::descriptors::no_casting Any other descriptors will be ignored.

Performance semantics:

1. Problem size N: $P * sizeof(IOType)$
2. local work: $N * Operator$;
3. transferred bytes: N ;
4. BSP cost: $Ng + N * Operator + l$;

9.10.2.2 broadcast() [1/2]

```
static RC broadcast (
            IOType & inout,
            const size_t root = 0 )  [inline], [static]
```

Schedules a broadcast operation of a single object of type IOType per process.

The broadcast shall be complete by the end of the call. This is a collective graphBLAS operation. The BSP costs are as for the PlatformBSP broadcast.

Template Parameters

IOType	The type of the to-be broadcast value.

Parameters

in,out	*inout*	On input at process *root:* the value to be broadcast. On input at non-root processes↩ : initial values are ignored. On output at process *root:* the input value remains unchanged. On output at non-root processes↩ : the same value held at process ID *root*.

Parameters

in	*root*	The user process which is to send out the given input value *inout* so that it becomes available at all *P* user processes. This value must be larger or equal to zero and must be smaller than the total number of user processes *P*.

Returns

SUCCESS On the successful completion of this function.

ILLEGAL When *root* is larger or equal to *P*. If this code is returned, it shall be as though the call to this function had never occurred. return PANIC When the function fails and the library enters an undefined state.

Performance semantics

Backends should define performance semantics in terms of work and data movement, the latter both within and between user processes. Also the number of synchronisations between user processes must be quantified.

Backends furthermore must indicate whether system calls may occur during a call to this primitive, indicate whether additional dynamic may be allocated (and if so, when it is freed), and quantify the required work space.

9.10.2.3 broadcast() [2/2]

```
static RC broadcast (
              IOType * inout,
              const size_t size,
              const size_t root = 0 )  [inline], [static]
```

Broadcast on an array of *IOType*.

The above documentation applies with *size* times `sizeof(IOType)` substituted in.

9.10.2.4 reduce()

```
static RC reduce (
              IOType & inout,
              const size_t root = 0,
              const Operator op = Operator() )  [inline], [static]
```

Schedules a reduce operation of a single object of type IOType per process.

The reduce shall be complete by the end of the call. This is a collective graphBLAS operation. The BSP costs are as for the PlatformBSP reduce.

Since this is a collective call, there are *P* values *inout* spread over all user processes. Let these values be denoted by x_s, with $s \in \{0, 1, \ldots, P-1\}$, such that x_s equals the argument *inout* on input at the user process with ID *s*. Let $\pi : \{0, 1, \ldots, P-1\} \to \{0, 1, \ldots, P-1\}$ be a bijection, some unknown permutation of the process ID. This permutation is must be fixed for any given combination of GraphBLAS implementation and value *P*. Let the binary operator *op* be denoted by $\odot$.

This function computes $\odot_{i=0}^{P-1} x_{\pi(i)}$ and writes the result to *inout* at the user process with ID *root*.

In summary, this the result is reproducible across different runs using the same input and *P*. Yet it does *not* mean that the order of addition is fixed.

Since each user process supplies but one value, there is no difference between a reduce-to-the-left versus a reduce-to-the-right (see grb::reducel and grb::reducer).

Template Parameters

descr	The GraphBLAS descriptor. Default is grb::descriptors::no_operation.
Operator	Which operator to use for reduction.
IOType	The type of the to-be reduced value.

Parameters

`in,out`	*inout*	On input↩ : the value at the calling process to be reduced. On output at process *root:* the reduced value. On output as non-root processes↩ : same value as on input.
`in`	*op*	The associative operator to reduce by.

Parameters

in	*root*	Which process should hold the reduced value. This number must be larger or equal to zero, and must be strictly smaller than the number of user processes *P*.

Returns

SUCCESS When the function completes successfully.

ILLEGAL When root is larger or equal than *P*. When this code is returned, the state of the GraphBLAS shall be as though this call was never made.

PANIC When an unmitigable error within the GraphBLAS occurs. Upon returning this error, the GraphBLAS enters an undefined state.

Note

If *op* is commutative, the implementation free to employ a different allreduce algorithm, as long as the performance semantics are documented so that its cost can be quantified.

Performance semantics:

1. Problem size N: $P * sizeof(IOType)$
2. local work: $N * Operator$;
3. transferred bytes: N ;
4. BSP cost: $Ng + N * Operator + l$;

The documentation for this class was generated from the following file:

- collectives.hpp

9.11 ConnectedComponents< VertexIDType > Struct Template Reference

A vertex-centric Connected Components algorithm.

`#include <pregel_connected_components.hpp>`

Classes

- struct Data

  This vertex-centric Connected Components algorithm does not require any algorithm parameters.

Static Public Member Functions

- template<typename PregelType >
  static grb::RC execute (grb::interfaces::Pregel< PregelType > &pregel, grb::Vector< VertexIDType > &group_ids, const size_t max_steps=0, size_t *const steps_taken=nullptr)

  A convenience function that, given a Pregel instance, executes the program.
- static void program (VertexIDType &current_max_ID, const VertexIDType &incoming_message, VertexIDType &outgoing_message, const Data &parameters, grb::interfaces::PregelState &pregel)

  The vertex-centric program for computing connected components.

9.11.1 Detailed Description

template<typename VertexIDType>
struct grb::algorithms::pregel::ConnectedComponents< VertexIDType >

A vertex-centric Connected Components algorithm.

Template Parameters

VertexIDType	A type large enough to assign an ID to each vertex in the graph the algorithm is to run on.

9.11.2 Member Function Documentation

9.11.2.1 execute()

```
static grb::RC execute (
            grb::interfaces::Pregel< PregelType > & pregel,
            grb::Vector< VertexIDType > & group_ids,
            const size_t max_steps = 0,
            size_t *const steps_taken = nullptr )  [inline], [static]
```

A convenience function that, given a Pregel instance, executes the program.

Parameters

in,out	*pregel*	A Pregel instance over which to execute the program.
out	*group_ids*	The ID of the component the corresponding vertex belongs to.
in	*max_steps*	A maximum number of rounds the program is allowed to run. If *0*, no maximum number of rounds will be in effect.

On succesful termination, the number of rounds is optionally written out:

Parameters

out	*steps_taken*	A pointer to where the number of rounds should be recorded. Will not be used if equal to nullptr.

9.11.2.2 program()

```
static void program (
            VertexIDType & current_max_ID,
            const VertexIDType & incoming_message,
            VertexIDType & outgoing_message,
            const Data & parameters,
            grb::interfaces::PregelState & pregel )  [inline], [static]
```

The vertex-centric program for computing connected components.

On termination, the number of individual IDs in *current_max_ID* signifies the number of components, while the value at each entry signifies which component the vertex corresponds to.

Parameters

in,out	*current_max_ID*	On input↩ : each entry is set to an unique ID, corresponding to a unique ID for each vertex. On output↩ : the ID of the component the corresponding vertex belongs to.
in	*incoming_message*	A buffer for incoming messages to a vertex program.
in	*outgoing_message*	A buffer for outgoing messages to a vertex program.

Parameters

in	*parameters*	Global algorithm parameters, currently an instance of an empty struct (no parameters).
in,out	*pregel*	The Pregel state the program may refer to.

This program 1) broadcasts its current ID to its neighbours, 2) checks if any received IDs are larger than the current ID, then 3a) if not, votes to halt; 3b) if yes, replaces the current ID with the received maximum. It is meant to be executed using a max monoid as message aggregator.

The documentation for this struct was generated from the following file:

- pregel_connected_components.hpp

9.12 Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType >::const_iterator Class Reference

A standard iterator for an ALP/GraphBLAS matrix.

`#include <matrix.hpp>`

Inherits iterator< std::forward_iterator_tag, std::pair< std::pair< const size_t, const size_t >, const D >, size_t >.

Public Member Functions

- bool operator!= (const const_iterator &other) const
- std::pair< const size_t, const D > operator* () const

  Dereferences the current position of this iterator.
- const_iterator & operator++ ()

  Advances the position of this iterator by one.
- bool operator== (const const_iterator &other) const

  Standard equals operator.

9.12.1 Detailed Description

template<typename D, enum Backend implementation, typename RowIndexType, typename ColIndexType, typename NonzeroIndexType>
class grb::Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType >::const_iterator

A standard iterator for an ALP/GraphBLAS matrix.

This iterator is used for data extraction only. Hence only this const version is specified.

Dereferencing an iterator of this type that is not in end position yields a pair (c, v). The value *v* is of type *D* and corresponds to the value of the dereferenced nonzero.

The value *c* is another pair (i, j). The values *i* and *j* are of type `size_t` and correspond to the coordinate of the dereferenced nonzero.

Note

'Pair' here corresponds to the regular `std::pair`.

Warning

Comparing two const iterators corresponding to different containers leads to undefined behaviour.

Advancing an iterator past the end iterator of the container it corresponds to, leads to undefined behaviour.

Modifying the contents of a container makes any use of any iterator derived from it incur invalid behaviour.

Note

These are standard limitations of STL iterators.

In terms of STL, the returned iterator is an *forward iterator*. Its performance semantics match that defined by the STL. Backends are encouraged to specify additional performance semantics as long as they do not conflict with those of a forward iterator.

Backends are allowed to return bi-directional or random access iterators instead of forward iterators.

9.12.2 Member Function Documentation

9.12.2.1 operator"!=()

```
bool operator!= (
            const const_iterator & other ) const  [inline]
```

Returns

The negation of operator==()

9.12.2.2 operator∗()

```
std::pair< const size_t, const D > operator* ( ) const  [inline]
```

Dereferences the current position of this iterator.

Returns

If this iterator is valid and not in end position, this returns an std::pair with in its first field the position of the nonzero value, and in its second field the value of the nonzero. The position of a nonzero is another std::pair with both the first and second field of type `size_t`.

Note

If this iterator is invalid or in end position, the result is undefined.

9.12.2.3 operator++()

```
const_iterator & operator++ ( )  [inline]
```

Advances the position of this iterator by one.

If the current position corresponds to the last element in the container, the new position of this iterator will be its end position.

If the current position of this iterator is already the end position, this iterator will become invalid; any use of invalid iterators will lead to undefined behaviour.

Returns

A reference to this iterator.

9.12.2.4 operator==()

```
bool operator== (
            const const_iterator & other ) const  [inline]
```

Standard equals operator.

Returns

Whether this iterator and the given *other* iterator are the same.

The documentation for this class was generated from the following file:

- matrix.hpp

9.13 Vector< D, implementation, C >::const_iterator Class Reference

A standard iterator for the Vector< D > class.

`#include <vector.hpp>`

Inherits iterator< std::forward_iterator_tag, std::pair< const size_t, const D >, size_t >.

Public Member Functions

- bool operator!= (const const_iterator &other) const
- std::pair< const size_t, const D > operator* () const

  Dereferences the current position of this iterator.
- const_iterator & operator++ ()

  Advances the position of this iterator by one.
- bool **operator==** (const const_iterator &other) const

  Standard equals operator.

9.13.1 Detailed Description

template<typename D, enum Backend implementation, typename C>
class grb::Vector< D, implementation, C >::const_iterator

A standard iterator for the Vector< D > class.

This iterator is used for data extraction only. Hence only this const version is supplied.

Warning

Comparing two const iterators corresponding to different containers leads to undefined behaviour.

Advancing an iterator past the end iterator of the container it corresponds to leads to undefined behaviour.

Modifying the contents of a container makes any use of any iterator derived from it incur invalid behaviour.

Note

These are standard limitations of STL iterators.

9.13.2 Member Function Documentation

9.13.2.1 operator"!=()

```
bool operator!= (
            const const_iterator & other ) const  [inline]
```

Returns

The negation of operator==().

9.13.2.2 operator*()

```
std::pair< const size_t, const D > operator* ( ) const  [inline]
```

Dereferences the current position of this iterator.

Returns

If this iterator is valid and not in end position, this returns a new std::pair with in its first field the position of the nonzero value, and in its second field the value of the nonzero.

Note

If this iterator is invalid or in end position, the result is, undefined.

9.13.2.3 operator++()

```
const_iterator & operator++ ( )  [inline]
```

Advances the position of this iterator by one.

If the current position corresponds to the last element in the container, the new position of this iterator will be its end position.

If the current position of this iterator is already the end position, this iterator will become invalid; any use of invalid iterators will lead to undefined behaviour.

Returns

A reference to this iterator.

The documentation for this class was generated from the following file:

- vector.hpp

9.14 ConnectedComponents< VertexIDType >::Data Struct Reference

This vertex-centric Connected Components algorithm does not require any algorithm parameters.

`#include <pregel_connected_components.hpp>`

9.14.1 Detailed Description

template<typename VertexIDType>
struct grb::algorithms::pregel::ConnectedComponents< VertexIDType >::Data

This vertex-centric Connected Components algorithm does not require any algorithm parameters.

The documentation for this struct was generated from the following file:

- pregel_connected_components.hpp

9.15 PageRank< IOType, localConverge >::Data Struct Reference

The algorithm parameters.

`#include <pregel_pagerank.hpp>`

Public Attributes

- IOType **alpha** = 0.15

  The probability of jumping to a random page instead of a linked page.
- IOType **tolerance** = 0.00001

  The local convergence criterion.

9.15.1 Detailed Description

template<typename IOType, bool localConverge>
struct grb::algorithms::pregel::PageRank< IOType, localConverge >::Data

The algorithm parameters.

The documentation for this struct was generated from the following file:

- pregel_pagerank.hpp

9.16 divide< D1, D2, D3, implementation > Class Template Reference

Numerical division of two numbers.

`#include <ops.hpp>`

Inherits Operator< internal::divide< D1, D1, D1, config::default_backend > >.

9.16.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::divide< D1, D2, D3, implementation >

Numerical division of two numbers.

Mathematical notation: $\odot(x, y) \rightarrow x/y$.

Note

This is the inverse to grb::operators::mul.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator/-functions available.

The documentation for this class was generated from the following file:

- ops.hpp

9.17 divide_reverse< D1, D2, D3, implementation > Class Template Reference

Reversed division of two numbers.

`#include <ops.hpp>`

Inherits Operator< internal::divide_reverse< D1, D1, D1, config::default_backend > >.

9.17.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::divide_reverse< D1, D2, D3, implementation >

Reversed division of two numbers.

Mathematical notation: $\odot(x, y) \rightarrow y/x$.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator/- functions available.

The documentation for this class was generated from the following file:

- ops.hpp

9.18 equal< D1, D2, D3, implementation > Class Template Reference

Operator which returns `true` if its inputs compare equal, and `false` otherwise.

`#include <ops.hpp>`

Inherits Operator< internal::equal< D1, D1, D1, config::default_backend > >.

9.18.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::equal< D1, D2, D3, implementation >

Operator which returns `true` if its inputs compare equal, and `false` otherwise.

Note

This operator is the inverse of grb::operators::not_equal.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator=- functions available.

The documentation for this class was generated from the following file:

- ops.hpp

9.19 equal_first< D1, D2, D3, implementation > Class Template Reference

Compares `std::pair` inputs taking the first entry in every pair as the comparison key, and returns `true` or `false` accordingly.

`#include <ops.hpp>`

Inherits Operator< internal::equal_first< D1, D1, D1, config::default_backend > >.

9.19.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::equal_first< D1, D2, D3, implementation >

Compares `std::pair` inputs taking the first entry in every pair as the comparison key, and returns `true` or `false` accordingly.

The input domains must both be `std::pair`.

Note

If the output type is not Boolean, the output is cast from Boolean to the output domain.

The output domain must hence be *castable* from `bool`.

The documentation for this class was generated from the following file:

- ops.hpp

9.20 geq< D1, D2, D3, implementation > Class Template Reference

This operation returns whether the left operand compares greater-than or equal to the right operand.

`#include <ops.hpp>`

Inherits Operator< internal::geq< D1, D1, D1, config::default_backend > >.

9.20.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::geq< D1, D2, D3, implementation >

This operation returns whether the left operand compares greater-than or equal to the right operand.

Mathematical notation: $\odot(x, y) \to x \geq y$.

The result is cast from `bool` to *D3*.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator>= overload available.

The documentation for this class was generated from the following file:

- ops.hpp

9.21 greater_than< D1, D2, D3, implementation > Class Template Reference

This operation returns whether the left operand compares greater-than the right operand.

`#include <ops.hpp>`

Inherits Operator< internal::gt< D1, D1, D1, config::default_backend > >.

9.21.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::greater_than< D1, D2, D3, implementation >

This operation returns whether the left operand compares greater-than the right operand.

Mathematical notation: $\odot(x, y) \to x > y$.

The result is cast from `bool` to *D3*.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator> overload available.

The documentation for this class was generated from the following file:

- ops.hpp

9.22 has_immutable_nonzeroes< T > Struct Template Reference

Used to inspect whether a given semiring has immutable nonzeroes under addition.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value** = false

  Whether T a semiring where nonzeroes are immutable.

9.22.1 Detailed Description

template<typename T>
struct grb::has_immutable_nonzeroes< T >

Used to inspect whether a given semiring has immutable nonzeroes under addition.

Template Parameters

T	The semiring to inspect.

An example of a monoid with an immutable identity is the logical OR, grb::operators::logical_or.

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.23 IMPLEMENTATION< backend > Class Template Reference

Collects a series of implementation choices corresponding to some given *backend*.

`#include <config.hpp>`

Static Public Member Functions

- static constexpr ALLOC_MODE defaultAllocMode ()
  Defines how private memory regions are allocated.
- static constexpr ALLOC_MODE sharedAllocMode ()
  Defines how shared memory regions are allocated.

9.23.1 Detailed Description

template<grb::Backend backend = default_backend>
class grb::config::IMPLEMENTATION< backend >

Collects a series of implementation choices corresponding to some given *backend*.

These implementation choices are useful for *compositional* backends; i.e., backends that rely on a nested sub-backend for functionality. To facilitate composability, backends are required to provide the functions specified herein.

Note

An example are the grb::BSP1D and grb::hybrid backends, that both share the exact same code, relying on either the grb::reference or the grb::reference_omp backend, respectively.

The user documentation does not list all required fields; for a complete overview, see the developer documentation instead.

The default class declaration is declared empty to ensure no one backend implicitly relies on global defaults. Every backend therefore must specialise this class and implement the specified functions.

Warning

Portable ALP user code does not rely on the implementation details gathered in this class.

Note

For properties of a backend that may (also) affect ALP user code, see grb::Properties.

The user documentation only documents the settings that could be useful to modify.

Warning

Modifying the documented functions should be done with care.

Any such modifications typically requires rebuilding the ALP library itself.

Note

For viewing all implementation choices, please see the developer documentation.

The documentation for this class was generated from the following file:

- base/config.hpp

9.24 IMPLEMENTATION< BSP1D > Class Reference

This class collects configuration parameters that are specific to the grb::BSP1D and grb::hybrid backends.

`#include <config.hpp>`

Static Public Member Functions

- static constexpr ALLOC_MODE defaultAllocMode ()
- static grb::config::ALLOC_MODE sharedAllocMode () noexcept

9.24.1 Detailed Description

This class collects configuration parameters that are specific to the grb::BSP1D and grb::hybrid backends.

Note

The full set of implementation details are only visible within the developer documentation.

9.24.2 Member Function Documentation

9.24.2.1 defaultAllocMode()

```
static constexpr ALLOC_MODE defaultAllocMode ( )  [inline], [static], [constexpr]
```

Returns

The default allocation strategy for private memory segments.

9.24.2.2 sharedAllocMode()

```
static grb::config::ALLOC_MODE sharedAllocMode ( )  [static], [noexcept]
```

Returns

The default allocation strategy for shared memory regions.

By default, for the BSP1D backend, a shared memory-segment should use interleaved alloc only if is running one process per compute node. This implies a run-time component to this function, which is why for this backend this function is *not* `constexpr`.

Warning

This function does assume that the number of processes does not change over the life time of an ALP context.

Note

While the above may seem a reasonably safe assumption, the use of the launcher in MANUAL mode may, in fact, make this a realistic issue that could be encountered. In such cases the deduction should be re-initiated. If you encounter this problem, please report it so that such a fix can be implemented.

The documentation for this class was generated from the following file:

- bsp1d/config.hpp

9.25 IMPLEMENTATION< nonblocking > Class Reference

Implementation-dependent configuration parameters for the *nonblocking* backend.

`#include <config.hpp>`

Static Public Member Functions

- static constexpr ALLOC_MODE defaultAllocMode ()

  A private memory segment shall never be accessed by threads other than the thread who allocates it.
- static constexpr ALLOC_MODE **sharedAllocMode** ()

  For the nonblocking backend, a shared memory-segment should use interleaved alloc so that any thread has uniform access on average.

9.25.1 Detailed Description

Implementation-dependent configuration parameters for the *nonblocking* backend.

Note

The user documentation only specifies the fields that under some circumstances may benefit from a user adapting it. For viewing all fields, please see the developer documentation.

Adapting the fields should be done with care and may require re-compilation and re-installation of the ALP framework.

See also

grb::config::IMPLEMENTATION

9.25.2 Member Function Documentation

9.25.2.1 defaultAllocMode()

```
static constexpr ALLOC_MODE defaultAllocMode ( )  [inline], [static], [constexpr]
```

A private memory segment shall never be accessed by threads other than the thread who allocates it.

Therefore we choose aligned mode here.

The documentation for this class was generated from the following file:

- nonblocking/config.hpp

9.26 IMPLEMENTATION< reference > Class Reference

This class collects configuration parameters that are specific to the grb::reference backend.

```
#include <config.hpp>
```

Static Public Member Functions

- static constexpr ALLOC_MODE **defaultAllocMode** ()

  How to allocate private memory segments.
- static constexpr ALLOC_MODE **sharedAllocMode** ()

  How to allocate shared memory segments.

9.26.1 Detailed Description

This class collects configuration parameters that are specific to the grb::reference backend.

It details both configurations that could be modified by end users, as well as configurations that are sensible only to ALP developers; the full specification hence is only available within the developer documentation.

The documentation for this class was generated from the following file:

- reference/config.hpp

9.27 IMPLEMENTATION< reference_omp > Class Reference

This class collects configuration parameters that are specific to the grb::reference_omp backend.

`#include <config.hpp>`

Static Public Member Functions

- static constexpr ALLOC_MODE defaultAllocMode ()

  A private memory segment shall never be accessed by threads other than the thread who allocates it.
- static constexpr ALLOC_MODE **sharedAllocMode** ()

  For the reference_omp backend, a shared memory-segment should use interleaved alloc so that any thread has uniform access on average.

9.27.1 Detailed Description

This class collects configuration parameters that are specific to the grb::reference_omp backend.

It details both configurations that could be modified by end users, as well as configurations that are sensible only to ALP developers; the full specification hence is only available within the developer documentation.

9.27.2 Member Function Documentation

9.27.2.1 defaultAllocMode()

`static constexpr ALLOC_MODE defaultAllocMode ( ) [inline], [static], [constexpr]`

A private memory segment shall never be accessed by threads other than the thread who allocates it.

Therefore we choose aligned mode here.

The documentation for this class was generated from the following file:

- reference/config.hpp

9.28 infinity< D > Class Template Reference

Standard identity for the minimum operator.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.28.1 Detailed Description

template<typename D>
class grb::identities::infinity< D >

Standard identity for the minimum operator.

9.28.2 Member Function Documentation

9.28.2.1 value()

`static constexpr D value ( )  [inline], [static], [constexpr]`

Template Parameters

D	The domain of the value to return.

Returns

The identity under the standard min operator (i.e., 'infinity'), of type *D*.

The documentation for this class was generated from the following file:

- identities.hpp

9.29 is_associative< T, typename > Struct Template Reference

Used to inspect whether a given operator or monoid is associative.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value** = false

  Whether T is associative.

9.29.1 Detailed Description

template<typename T, typename = void>
struct grb::is_associative< T, typename >

Used to inspect whether a given operator or monoid is associative.

Note

Monoids are associative by definition, but this type traits is nonetheless defined for them so as to preserve symmetry in the API; see, e.g., grb::is_commutative or grb::is_idempotent.

Template Parameters

T	The operator or monoid to inspect.

An example of an associative operator is the logical or, grb::operators::logical_or.

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.30 is_commutative< T, typename > Struct Template Reference

Used to inspect whether a given operator or monoid is commutative.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value** = false

  Whether T is commutative.

9.30.1 Detailed Description

template<typename T, typename = void>
struct grb::is_commutative< T, typename >

Used to inspect whether a given operator or monoid is commutative.

Template Parameters

T	The operator or monoid to inspect.

An example of a commutative operator is numerical addition, grb::operators::add.

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.31 is_container< T > Struct Template Reference

Used to inspect whether a given type is an ALP/GraphBLAS container.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value** = false
  Whether T is an ALP/GraphBLAS container.

9.31.1 Detailed Description

template<typename T>
struct grb::is_container< T >

Used to inspect whether a given type is an ALP/GraphBLAS container.

Template Parameters

T	The type to inspect.

There are only two ALP/GraphBLAS containers:

1. grb::Vector, and
2. grb::Matrix.

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.32 is_idempotent< T, typename > Struct Template Reference

Used to inspect whether a given operator or monoid is idempotent.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value** = false
  Wheter T is idempotent.

9.32.1 Detailed Description

template<typename T, typename = void>
struct grb::is_idempotent< T, typename >

Used to inspect whether a given operator or monoid is idempotent.

Template Parameters

T	The operator or monoid to inspect.

An example of an idempotent operator is the logical OR, grb::operators::logical_or.

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.33 is_monoid< T > Struct Template Reference

Used to inspect whether a given type is an ALP monoid.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value** = false
  Whether T is an ALP monoid.

9.33.1 Detailed Description

template<typename T>
struct grb::is_monoid< T >

Used to inspect whether a given type is an ALP monoid.

Template Parameters

T	The type to inspect.

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.34 is_object< T > Struct Template Reference

Used to inspect whether a given type is an ALP/GraphBLAS object.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value**

  Whether the given time is an ALP/GraphBLAS object.

9.34.1 Detailed Description

template<typename T>
struct grb::is_object< T >

Used to inspect whether a given type is an ALP/GraphBLAS object.

Template Parameters

T	The type to inspect.

An ALP/GraphBLAS object is either an ALP/GraphBLAS container or an ALP semiring, monoid, or operator.

See also

grb::is_monoid
grb::is_semiring
grb::is_operator
grb::is_container

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.35 is_operator< T > Struct Template Reference

Used to inspect whether a given type is an ALP operator.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value** = false
  Whether T is an ALP operator.

9.35.1 Detailed Description

template<typename T>
struct grb::is_operator< T >

Used to inspect whether a given type is an ALP operator.

Template Parameters

T	The type to inspect.

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.36 is_semiring< T > Struct Template Reference

Used to inspect whether a given type is an ALP semiring.

`#include <type_traits.hpp>`

Static Public Attributes

- static const constexpr bool **value** = false
  Whether T is an ALP semiring.

9.36.1 Detailed Description

template<typename T>
struct grb::is_semiring< T >

Used to inspect whether a given type is an ALP semiring.

Template Parameters

T	The type to inspect.

The documentation for this struct was generated from the following file:

- type_traits.hpp

9.37 Launcher< mode, backend > Class Template Reference

A group of user processes that together execute ALP programs.

`#include <exec.hpp>`

Public Member Functions

- Launcher (const size_t process_id=0, const size_t nprocs=1, const std::string hostname="localhost", const std::string port="0")

  Constructs a new grb::Launcher.
- template<typename T , typename U >
  RC exec (void(*alp_program)(const T &, U &), const T &data_in, U &data_out, const bool broadcast=false) const

  Executes a given ALP program using the user processes encapsulated by this launcher group.
- template<typename U >
  RC exec (void(*alp_program)(const void *, const size_t, U &), const void *data_in, const size_t in_size, U &data_out, const bool broadcast=false) const

  Executes a given ALP program using the user processes encapsulated by this launcher group.

Static Public Member Functions

- static RC finalize ()

  Releases all ALP resources.

9.37.1 Detailed Description

template<enum EXEC_MODE mode, enum Backend backend>
class grb::Launcher< mode, backend >

A group of user processes that together execute ALP programs.

Allows an application to run any ALP program. Input data may be passed through a user-defined type. Output data will be retrieved via the same type.

For backends that support multiple user processes, the caller may explicitly set the process ID and total number of user processes.

The intended use is to 'just call' the exec function, which should be accepted by any backend.

Template Parameters

mode	Which EXEC_MODE the Launcher should adhere to.
backend	Which backend is to be used.

9.37.2 Constructor & Destructor Documentation

9.37.2.1 Launcher()

```
Launcher (
            const size_t process_id = 0,
            const size_t nprocs = 1,
            const std::string hostname = "localhost",
            const std::string port = "0" )  [inline]
```

Constructs a new grb::Launcher.

This constructor is a collective call; all *nprocs* processes that form a single launcher group must make a simultaneous call to this constructor.

There is an implementation-defined time-out for the creation of a launcher group.

Parameters

Parameters

in	*process_id*	The user process ID of the calling process. The value must be larger or equal to 0. This value must be strictly smaller than *nprocs*. This value must be unique to the calling process within this collective call across *all nprocs* user processes. This number *must* be strictly smaller than *nprocs*. Optional: the default is 0.

Parameters

in	*nprocs*	The total number of user processes making a collective call to this function. Optional: the default is 1.
in	*hostname*	The hostname of one of the user processes. Optional: the default is 'localhost'.

Parameters

in	*port*	A free port number at *hostname*. This port will be used for TCP connections to *hostname* if and only if *nprocs* is larger than one. Optional: the default value is '0'.

Exceptions

invalid_argument	If *nprocs* is zero.
invalid_argument	If *process_id* is greater than or equal to *nprocs*.

Note

An implementation or backend may define further constraints on the input arguments, such as, obviously, on *hostname* and *port*, but also on *nprocs* and, as a result, on *process_id*.

The most obvious is that backends supporting only one user process must not accept *nprocs* larger than 1.

All aforementioned default values shall always be legal.

9.37.3 Member Function Documentation

9.37.3.1 exec() [1/2]

```
RC exec (
            void(*)(const T &, U &) alp_program,
            const T & data_in,
            U & data_out,
            const bool broadcast = false ) const  [inline]
```

Executes a given ALP program using the user processes encapsulated by this launcher group.

Calling this function, depending on whether the automatic or manual/MPI mode was selected, will either *spawn* the maximum number of available user processes and *then* execute the given program, *or* it will employ the given processes that are managed by the user application and used to construct this launcher instance to execute the given *alp_program*.

This is a collective function call– all processes in the launcher group must make a simultaneous call to this function and must do so using consistent arguments.

Template Parameters

T	The type of the data to pass to the ALP program as input.
U	The type of the output data to pass back to the caller.

Parameters

in	*alp_program*	The user program to be executed.
in	*data_in*	Input data of user-defined type *T*.

When in automatic mode and *broadcast* is `false`, the data will only be available at user process with ID 0. When in automatic mode and *broadcast* is `true`, the data will be available at all user processes. When in manual mode, the data will be available to this user process only, with "this process" corresponding to the process that calls this function.

Parameters

out	*data_out*	Output data of user-defined type *U*. The output data should be available at user process with ID zero.
in	*broadcast*	Whether the input should be broadcast from user process 0 to all other user processes. Optional; the default value is *false*.

Returns

grb::SUCCESS If the execution proceeded as intended.

grb::PANIC If an unrecoverable error was encountered while attempting to execute, attempting to terminate, or while executing, the given program.

Warning

Even if grb::SUCCESS is returned, an algorithm may fail to achieve its intended result– for example, an iterative solver may fail to converge. A good programming pattern has that *U* either a) is an error code for the algorithm used (e.g., grb::RC), or b) that *U* contains such an error code.

9.37.3.2 exec() [2/2]

```
RC exec (
            void(*)(const void *, const size_t, U &) alp_program,
            const void * data_in,
            const size_t in_size,
            U & data_out,
            const bool broadcast = false ) const  [inline]
```

Executes a given ALP program using the user processes encapsulated by this launcher group.

This variant of exec has that *data_in* is of a variable byte size, instead of a fixed POD type. If *broadcast* is `true` and the launcher is instantiated using the grb::AUTOMATIC mode, all bytes are broadcast to all user processes.

Parameters

in	*alp_program*	The user program to be executed.
in	*data_in*	Pointer to raw input byte data.
in	*in_size*	The number of bytes the input data consists of.
out	*data_out*	Output data of user-defined type *U*. The output data should be available at user process with ID zero.

Parameters

in	*broadcast*	Whether the input should be broadcast from user process 0 to all other user processes. Optional; the default value is *false*.

Returns

grb::SUCCESS If the execution proceeded as intended.

grb::PANIC If an unrecoverable error was encountered while attempting to execute, attempting to terminate, or while executing, the given program.

For more details, see the other version of this function.

9.37.3.3 finalize()

```
static RC finalize ( )  [inline], [static]
```

Releases all ALP resources.

After a call to this function, no further ALP programs may launched using the grb::Launcher and grb::Benchmarker. Also the use of grb::init and grb::finalize will no longer be accepted.

Warning

grb::init and grb::finalize are deprecated.

After a call to this function, the only way to once again run ALP programs is to use the grb::Launcher from a new process.

Warning

Therefore, use this function with care and preferably only just before exiting the process.

A well-behaving program calls this function, or grb::Benchmarker::finalize, exactly once before its process terminates, or just after the guaranteed last invocation of an ALP program.

Returns

grb::SUCCESS The resources have successfully and permanently been released.

grb::PANIC An unrecoverable error has been encountered and the user program is encouraged to exit as quickly as possible. The state of the ALP library has become undefined and should no longer be used.

Note

In the terminology of the Message Passing Interface (MPI), this function is the ALP equivalent of the `MPI_Finalize()`.

In grb::AUTOMATIC mode when using a parallel backend that uses MPI to auto-parallelise the ALP computations, MPI is never explicitly exposed to the user application. This use case necessitates the specification of this function.

Thus, and in particular, an ALP program launched in grb::AUTOMATIC mode while using the grb::BSP1D or the grb::hybrid backends with ALP compiled using LPF that in turn is configured to use an MPI-based engine, should make sure to call this function before program exit.

An application that launches ALP programs in grb::FROM_MPI mode must still call this function, even though a proper such application makes its own call to `MPI_Finalize()`. This does *not* induce improper behaviour since calling this function using a launcher instance in grb::FROM_MPI mode translates, from an MPI perspective, to a no-op.

The documentation for this class was generated from the following file:

- exec.hpp

9.38 left_assign< D1, D2, D3, implementation > Class Template Reference

This operator discards all right-hand side input and simply copies the left-hand side input to the output variable.

`#include <ops.hpp>`

Inherits Operator< internal::left_assign< D1, D1, D1, config::default_backend > >.

9.38.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::left_assign< D1, D2, D3, implementation >

This operator discards all right-hand side input and simply copies the left-hand side input to the output variable.

It exposes the complete interface detailed in grb::operators::internal::Operator. This operator can be passed to any GraphBLAS function or object constructor.

Mathematical notation: $\odot(x, y) \rightarrow x$.

Template Parameters

D1	The left-hand side input domain.
D2	The right-hand side input domain.
D3	The output domain.

The documentation for this class was generated from the following file:

- ops.hpp

9.39 left_assign_if< D1, D2, D3, implementation > Class Template Reference

This operator assigns the left-hand input if the right-hand input evaluates `true`.

`#include <ops.hpp>`

Inherits Operator< internal::left_assign_if< D1, D1, D1, config::default_backend > >.

9.39.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::left_assign_if< D1, D2, D3, implementation >

This operator assigns the left-hand input if the right-hand input evaluates `true`.

If the right-hand input does not evaluate `true`, then the output field is unmodified.

Warning

Therefore, this operator may propagate the use of uninitialised values if not used with care. Ensuring its use with in-place primitives is recommended.

The documentation for this class was generated from the following file:

- ops.hpp

9.40 leq< D1, D2, D3, implementation > Class Template Reference

This operation returns whether the left operand compares less-than or equal to the right operand.

`#include <ops.hpp>`

Inherits Operator< internal::leq< D1, D1, D1, config::default_backend > >.

9.40.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::leq< D1, D2, D3, implementation >

This operation returns whether the left operand compares less-than or equal to the right operand.

Mathematical notation: $\odot(x, y) \to x \leq y$.

The result is cast from `bool` to *D3*.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator<= overload available.

The documentation for this class was generated from the following file:

- ops.hpp

9.41 less_than< D1, D2, D3, implementation > Class Template Reference

This operation returns whether the left operand compares less-than the right operand.

`#include <ops.hpp>`

Inherits Operator< internal::lt< D1, D1, D1, config::default_backend > >.

9.41.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::less_than< D1, D2, D3, implementation >

This operation returns whether the left operand compares less-than the right operand.

Mathematical notation: $\odot(x, y) \to x < y$.

The result is cast from `bool` to *D3*.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator< overload available.

The documentation for this class was generated from the following file:

- ops.hpp

9.42 logical_and< D1, D2, D3, implementation > Class Template Reference

The logical and.

`#include <ops.hpp>`

Inherits Operator< internal::logical_and< D1, D1, D1, config::default_backend > >.

9.42.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::logical_and< D1, D2, D3, implementation >

The logical and.

It returns `true` when both of its inputs evaluate `true`, and returns `false` otherwise.

If the output domain is not Boolean, then the returned value is `true` or `false` cast to the output domain.

Warning

Thus both input domains and the output domain must be *castable* to `bool`.

The documentation for this class was generated from the following file:

- ops.hpp

9.43 logical_false< D > Class Template Reference

Standard identitity for the logical or operator.

`#include <identities.hpp>`

Static Public Member Functions

- static const constexpr D value ()

9.43.1 Detailed Description

template<typename D>
class grb::identities::logical_false< D >

Standard identitity for the logical or operator.

See also

operators::logical_or.

9.43.2 Member Function Documentation

9.43.2.1 value()

```
static const constexpr D value ( )  [inline], [static], [constexpr]
```

Template Parameters

D	The domain of the value to return.

Returns

The identity under the standard logical OR operator, i.e., *false*.

The documentation for this class was generated from the following file:

- identities.hpp

9.44 logical_or< D1, D2, D3, implementation > Class Template Reference

The logical or.

`#include <ops.hpp>`

Inherits Operator< internal::logical_or< D1, D1, D1, config::default_backend > >.

9.44.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::logical_or< D1, D2, D3, implementation >

The logical or.

It returns `true` whenever any of its inputs evaluate `true`, and returns `false` otherwise.

If the output domain is not Boolean, then the returned value is `true` or `false` cast to the output domain.

Warning

Thus both input domains and the output domain must be *castable* to `bool`.

The documentation for this class was generated from the following file:

- ops.hpp

9.45 logical_true< D > Class Template Reference

Standard identity for the logical AND operator.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.45.1 Detailed Description

template<typename D>
class grb::identities::logical_true< D >

Standard identity for the logical AND operator.

See also

operators::logical_and.

9.45.2 Member Function Documentation

9.45.2.1 value()

```
static constexpr D value ( )  [inline], [static], [constexpr]
```

Template Parameters

D	The domain of the value to return.

Returns

The identity under the standard logical AND operator, i.e., *true*.

The documentation for this class was generated from the following file:

- identities.hpp

9.46 Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType > Class Template Reference

An ALP/GraphBLAS matrix.

```
#include <matrix.hpp>
```

Classes

- class const_iterator

  A standard iterator for an ALP/GraphBLAS matrix.

Public Types

- typedef Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType > **self_type**

  The type of this container.
- typedef D **value_type**

  The value type of elements stored in this matrix.

Public Member Functions

- Matrix (const Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType > &other)

  Copy constructor.
- Matrix (const size_t rows, const size_t columns)

  ALP/GraphBLAS matrix constructor that sets a default initial capacity.
- Matrix (const size_t rows, const size_t columns, const size_t nz)

  ALP/GraphBLAS matrix constructor that sets an initial capacity.
- Matrix (self_type &&other)

  Move constructor.
- ~Matrix ()

  Matrix destructor.
- const_iterator begin () const

  Same as cbegin().
- const_iterator cbegin () const

  Provides the only mechanism to extract data from a GraphBLAS matrix.
- const_iterator cend () const

  Indicates the end to the elements in this container.
- const_iterator end () const

  Same as cend().
- self_type & operator= (self_type &&other) noexcept

  Move-assignment.

9.46.1 Detailed Description

template<typename D, enum Backend implementation, typename RowIndexType, typename ColIndexType, typename NonzeroIndexType>
class grb::Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType >

An ALP/GraphBLAS matrix.

This is an opaque data type that implements the below constructors, member functions, and destructors.

Template Parameters

D	The type of a nonzero element.

The given type *D* shall not be an ALP/GraphBLAS object.

Template Parameters

implementation	Allows multiple backends to implement different versions of this data type.

Warning

Creating a grb::Matrix of other ALP/GraphBLAS types is not allowed.

9.46.2 Constructor & Destructor Documentation

9.46.2.1 Matrix() [1/4]

```
Matrix (
            const size_t rows,
            const size_t columns,
            const size_t nz )  [inline]
```

ALP/GraphBLAS matrix constructor that sets an initial capacity.

Parameters

in	*rows*	The number of rows of the matrix to be instantiated.
in	*columns*	The number of columns of the matrix to be instantiated.
in	*nz*	The minimum initial capacity of the matrix to be instantiated.

After successful construction, the resulting matrix has a capacity of *at least nz* nonzeroes. If either *rows* or *columns* is 0, then the capacity may instead be 0 as well.

On errors such as out-of-memory, this constructor may throw exceptions.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

Warning

Avoid the use of this constructor within performance critical code sections.

9.46.2.2 Matrix() [2/4]

```
Matrix (
            const size_t rows,
            const size_t columns )  [inline]
```

ALP/GraphBLAS matrix constructor that sets a default initial capacity.

Parameters

in	*rows*	The number of rows in the new matrix.
in	*columns*	The number of columns in the new matrix.

The default capacity is the maximum of *rows* and *columns*.

On errors such as out-of-memory, this constructor may throw exceptions.

For the full specification, please see the full constructor signature.

Warning

Avoid the use of this constructor within performance critical code sections.

9.46.2.3 Matrix() [3/4]

```
Matrix (
            const Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType > &
other )  [inline]
```

Copy constructor.

Parameters

in	*other*	The matrix to copy.

This performs a deep copy; a new matrix is allocated with the same (or larger) capacity as *other*, after which the contents of *other* are copied into the new instance.

The use of this constructor is semantically the same as:

```
grb::Matrix< T > newMatrix(
    grb::nrows( other ), grb::ncols( other ),
    grb::capacity( other )
);
grb::set( newMatrix, other );
```

(Under the condition that all calls are successful.)

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

Warning

Avoid the use of this constructor within performance critical code sections.

9.46.2.4 Matrix() [4/4]

```
Matrix (
            self_type && other )  [inline]
```

Move constructor.

This will take over the resources of the given *other* matrix, invalidating the contents of *other* while its contents are now moved into this instance instead.

Parameters

in	*other*	The matrix to move to this new instance.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

9.46.2.5 ~Matrix()

```
~Matrix ( )  [inline]
```

Matrix destructor.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

Warning

Avoid calling destructors from within performance critical code sections.

9.46.3 Member Function Documentation

9.46.3.1 begin()

```
const_iterator begin ( ) const  [inline]
```

Same as cbegin().

Since iterators are only supplied as a data extraction mechanism, there is no overloaded version of this function that returns a non-const iterator.

9.46.3.2 cbegin()

```
const_iterator cbegin ( ) const  [inline]
```

Provides the only mechanism to extract data from a GraphBLAS matrix.

The order in which nonzero elements are returned is undefined.

Returns

An iterator pointing to the first element of this matrix, if any; *or* an iterator in end position if this vector contains no nonzeroes.

Note

An 'iterator in end position' compares equal to the const_iterator returned by cend().

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

Note

This function may make use of a const_iterator that is buffered, hence possibly causing its implicitly called constructor to allocate dynamic memory.

Warning

Avoid the use of this function within performance critical code sections.

9.46.3.3 cend()

```
const_iterator cend ( ) const  [inline]
```

Indicates the end to the elements in this container.

Returns

An iterator at the end position of this container.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

Note

Even if cbegin() returns a buffered const_iterator that may require dynamic memory allocation and additional data movement, this specification disallows the same to happen for the construction of an iterator in end position.

Warning

Avoid the use of this function within performance critical code sections.

9.46.3.4 end()

```
const_iterator end ( ) const  [inline]
```

Same as cend().

Since iterators are only supplied as a data extraction mechanism, there is no overloaded version of this function that returns a non-const iterator.

9.46.3.5 operator=()

```
self_type & operator= (
            self_type && other )  [inline], [noexcept]
```

Move-assignment.

This will take over the resources of the given *other* matrix, invalidating the contents of *other* while its contents are now moved into this instance instead.

This will destroy any current contents in this container.

Parameters

in	*other*	The matrix contents to move into this instance.

Performance semantics.

Each backend must define performance semantics for this primitive.

See also

Performance Semantics

The documentation for this class was generated from the following file:

- matrix.hpp

9.47 max< D1, D2, D3, implementation > Class Template Reference

This operator takes the maximum of the two input parameters and writes the result to the output variable.

`#include <ops.hpp>`

Inherits Operator< internal::max< D1, D1, D1, config::default_backend > >.

9.47.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::max< D1, D2, D3, implementation >

This operator takes the maximum of the two input parameters and writes the result to the output variable.

It exposes the complete interface detailed in grb::operators::internal::Operator. This operator can be passed to any GraphBLAS function or object constructor.

Mathematical notation: $\max(x, y) \rightarrow \begin{cases} x \text{ if } x > y \\ y \text{ otherwise} \end{cases}$.

Template Parameters

D1	The left-hand side input domain.
D2	The right-hand side input domain.
D3	The output domain.

Warning

This operator expects objects with a partial ordering defined on and between elements of types *D1*, *D2*, and *D3*.

The documentation for this class was generated from the following file:

- ops.hpp

9.48 MEMORY Class Reference

Memory configuration parameters.

`#include <config.hpp>`

Static Public Member Functions

- static constexpr size_t big_memory ()
- static constexpr size_t l1_cache_size ()

9.48.1 Detailed Description

Memory configuration parameters.

The documentation for this class was generated from the following file:

- base/config.hpp

9.49 min< D1, D2, D3, implementation > Class Template Reference

This operator takes the minimum of the two input parameters and writes the result to the output variable.

`#include <ops.hpp>`

Inherits Operator< internal::min< D1, D1, D1, config::default_backend > >.

9.49.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::min< D1, D2, D3, implementation >

This operator takes the minimum of the two input parameters and writes the result to the output variable.

It exposes the complete interface detailed in grb::operators::internal::Operator. This operator can be passed to any GraphBLAS function or object constructor.

Mathematical notation: $\max(x, y) \rightarrow \begin{cases} x \text{ if } x < y \\ y \text{ otherwise} \end{cases}$.

Template Parameters

D1	The left-hand side input domain.
D2	The right-hand side input domain.
D3	The output domain.

Warning

This operator expects objects with a partial ordering defined on and between elements of types *D1*, *D2*, and *D3*.

The documentation for this class was generated from the following file:

- ops.hpp

9.50 Monoid< _OP, _ID > Class Template Reference

A generalised monoid.

`#include <monoid.hpp>`

Public Types

- typedef _OP::D1 **D1**

  The left-hand side input domain.
- typedef _OP::D2 **D2**

  The right-hand side input domain.
- typedef _OP::D3 **D3**

  The output domain.
- template<typename IdentityType >
  using **Identity** = _ID< IdentityType >

  The underlying identity.
- typedef _OP **Operator**

  The type of the underlying operator.

Public Member Functions

- Monoid ()

  Constructor that infers a default operator, given the operator type.
- template<typename D >
  constexpr D getIdentity () const

  Retrieves the identity corresponding to this monoid.
- Operator getOperator () const

  Retrieves the underlying operator.

9.50.1 Detailed Description

template<class _OP, template< typename > class _ID>
class grb::Monoid< _OP, _ID >

A generalised monoid.

Template Parameters

_OP	The monoid operator.
_ID	The monoid identity (the '0').

9.50.2 Constructor & Destructor Documentation

9.50.2.1 Monoid()

```
Monoid ( )  [inline]
```

Constructor that infers a default operator, given the operator type.

Useful for stateless operators.

9.50.3 Member Function Documentation

9.50.3.1 getIdentity()

```
constexpr D getIdentity ( ) const  [inline], [constexpr]
```

Retrieves the identity corresponding to this monoid.

The identity value will be cast to the requested domain.

Template Parameters

D	The requested domain of the identity.

Returns

The identity corresponding to this monoid, cast to the requested domain.

9.50.3.2 getOperator()

```
Operator getOperator ( ) const  [inline]
```

Retrieves the underlying operator.

Returns

The underlying operator. Any state is copied.

The documentation for this class was generated from the following file:

- monoid.hpp

9.51 mul< D1, D2, D3, implementation > Class Template Reference

This operator multiplies the two input parameters and writes the result to the output variable.

`#include <ops.hpp>`

Inherits Operator< internal::mul< D1, D1, D1, config::default_backend > >.

9.51.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::mul< D1, D2, D3, implementation >

This operator multiplies the two input parameters and writes the result to the output variable.

It exposes the complete interface detailed in grb::operators::internal::Operator. This operator can be passed to any GraphBLAS function or object constructor.

Mathematical notation: $\odot(x, y) \rightarrow x \cdot y$.

Template Parameters

D1	The left-hand side input domain.
D2	The right-hand side input domain.
D3	The output domain.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator∗-functions available.

The documentation for this class was generated from the following file:

- ops.hpp

9.52 negative_infinity< D > Class Template Reference

Standard identity for the maximum operator.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.52.1 Detailed Description

template<typename D>
class grb::identities::negative_infinity< D >

Standard identity for the maximum operator.

9.52.2 Member Function Documentation

9.52.2.1 value()

`static constexpr D value ( )  [inline], [static], [constexpr]`

Template Parameters

D	The domain of the value to return.

Returns

The identity under the standard max operator, i.e., 'minus infinity'.

The documentation for this class was generated from the following file:

- identities.hpp

9.53 not_equal< D1, D2, D3, implementation > Class Template Reference

Operator that returns `false` whenever its inputs compare equal, and `true` otherwise.

`#include <ops.hpp>`

Inherits Operator< internal::not_equal< D1, D1, D1, config::default_backend > >.

9.53.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::not_equal< D1, D2, D3, implementation >

Operator that returns `false` whenever its inputs compare equal, and `true` otherwise.

Note

This operator is the inverse of grb::operators::equal.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator=-functions available.

The documentation for this class was generated from the following file:

- ops.hpp

9.54 one< D > Class Template Reference

Standard identity for numerical multiplication.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.54.1 Detailed Description

template<typename D>
class grb::identities::one< D >

Standard identity for numerical multiplication.

9.54.2 Member Function Documentation

9.54.2.1 value()

```
static constexpr D value ( )  [inline], [static], [constexpr]
```

Template Parameters

D	The domain of the value to return.

Returns

The identity under standard multiplication (i.e., 'one').

The documentation for this class was generated from the following file:

- identities.hpp

9.55 PageRank< IOType, localConverge > Struct Template Reference

A Pregel-style PageRank-like algorithm.

```
#include <pregel_pagerank.hpp>
```

Classes

- struct Data

  The algorithm parameters.

Static Public Member Functions

- template<typename PregelType >
  static grb::RC execute (grb::interfaces::Pregel< PregelType > &pregel, grb::Vector< IOType > &scores, size_t &steps_taken, const Data &parameters=Data(), const size_t max_steps=0)

  A convenience function for launching a PageRank algorithm over a given Pregel instance.
- static void program (IOType &current_score, const IOType &incoming_message, IOType &outgoing_↩ message, const Data &parameters, grb::interfaces::PregelState &pregel)

  The vertex-centric PageRank-like program.

9.55.1 Detailed Description

template<typename IOType, bool localConverge>
struct grb::algorithms::pregel::PageRank< IOType, localConverge >

A Pregel-style PageRank-like algorithm.

This vertex-centric program does not correspond to the canonical PageRank algorithm by Brin and Page. In particular, it misses corrections for dangling nodes and does not perform convergence checks in any norm.

Template Parameters

IOType	The type of the PageRank scores (e.g., `double`).
localConverge	Whether vertices become inactive once their local scores have converged, or whether to terminate only when all v

9.55.2 Member Function Documentation

9.55.2.1 execute()

```
static grb::RC execute (
            grb::interfaces::Pregel< PregelType > & pregel,
            grb::Vector< IOType > & scores,
            size_t & steps_taken,
            const Data & parameters = Data(),
            const size_t max_steps = 0 )  [inline], [static]
```

A convenience function for launching a PageRank algorithm over a given Pregel instance.

Template Parameters

PregelType	The nonzero type of an edge in the Pregel instance.

This convenience function materialises the buffers expected to be passed into the Pregel instance, and selects the expected monoid for executing this program.

Warning

> In performance-critical code, one may want to pre-allocate the buffers instead of having this convenience function allocate those. In such cases, please call manually the Pregel execute function, i.e., grb::interfaces::Pregel< PregelType >::execute.

The following arguments are mandatory:

Parameters

in	*pregel*	The Pregel instance that this program should execute on.
out	*scores*	A vector that corresponds to the scores corresponding to each vertex. It must be of size equal to the number of vertices n in the *pregel* instance, and must have n capacity *and* values. The initial contents are ignored by this algorithm.

Parameters

out	*steps_taken*	How many rounds the program took until termination.

The following arguments are optional:

Parameters

in	*parameters*	The algorithm parameters. If not given, default values will be substituted.
in	*max_steps*	The maximum number of rounds this program may take. If not given, the number of rounds will be unlimited.

9.55.2.2 program()

```
static void program (
            IOType & current_score,
```

```
const IOType & incoming_message,
IOType & outgoing_message,
const Data & parameters,
grb::interfaces::PregelState & pregel )  [inline], [static]
```

The vertex-centric PageRank-like program.

Parameters

out	*current_score*	The current rank corresponding to this vertex.
in	*incoming_message*	Neighbour contributions to our score.
out	*outgoing_message*	The score contribution to send to our neighbours.
in	*parameters*	The algorithm parameters.
in,out	*pregel*	The state of the Pregel interface.

The Pregel program expects incoming messages to be aggregated using a plus monoid over elements of *IOType*.

The documentation for this struct was generated from the following file:

- pregel_pagerank.hpp

9.56 PinnedVector< IOType, implementation > Class Template Reference

Provides a mechanism to access ALP containers from outside of an ALP context.

`#include <pinnedvector.hpp>`

Public Member Functions

- PinnedVector ()

  Base constructor for this class.
- template<typename Coord >
  PinnedVector (const Vector< IOType, implementation, Coord > &vector, const IOMode mode)

  Pins the contents of a given vector.
- ~PinnedVector ()

  Destroys a grb::PinnedVector instance.
- size_t getNonzeroIndex (const size_t k) const noexcept

  Retrieves a nonzero index.
- IOType getNonzeroValue (const size_t k) const noexcept

  Direct access variation of the general getNonzeroValue function.
- template<typename OutputType >
  OutputType getNonzeroValue (const size_t k, const OutputType one=OutputType()) const noexcept

  Returns a requested nonzero of the pinned vector.
- size_t nonzeroes () const noexcept
- size_t size () const noexcept

9.56.1 Detailed Description

template<typename IOType, enum Backend implementation>
class grb::PinnedVector< IOType, implementation >

Provides a mechanism to access ALP containers from outside of an ALP context.

An instance of grb::PinnedVector caches a container's data and returns it to the user. The user can refer to the returned data until such time the instance of grb::PinnedVector is destroyed, regardless of whether a call to grb::finalize occurs, and regardless whether the ALP/GraphBLAS program executed through the grb::Launcher had already returned.

The original container may not be modified or any derived instance of *PinnedVector* shall become invalid.

Note

It would be strange if an ALP/GraphBLAS container a pinned vector is derived from persists– pinned vectors are designed to be used precisely when the original container no longer is in scope. Therefore this last remark on invalidation should not matter.

The grb::PinnedVector abstracts a read-only container of nonzeroes. A nonzero is a pair of indices and values. One may query for the number of nonzeroes and use

1. getNonzeroValue to retrieve a nonzero value, or
2. getNonzeroIndex to retrieve a nonzero index.

An instance of grb::PinnedVector cannot modify the underlying nonzero structure nor can it modify its values.

Note

A performant implementation in fact does *not* copy the container data, but provides a mechanism to access the underlying ALP memory whenever it is possible to do so. This memory should remain valid even after a call to grb::Launcher::exec has completed, and for as long as the grb::PinnedVector instance remains valid.

9.56.2 Constructor & Destructor Documentation

9.56.2.1 PinnedVector() [1/2]

```
PinnedVector (
            const Vector< IOType, implementation, Coord > & vector,
            const IOMode mode )  [inline]
```

Pins the contents of a given *vector*.

A successfully constructed grb::PinnedVector shall remain valid until it is destroyed, regardless of whether the ALP context in which the original *vector* appears has been destroyed.

Pinning may or may not require a memory copy, depending on the ALP implementation and backend. If it does not, then destroying this instance *may* result in memory deallocation. It only *must* result in deallocation if the pinned vector that did not require a memory copy happens to be the last remaining reference to the original *vector*.

If one user process calls this constructor, *all* user processes must do so and with the same arguments– this is a collective call.

All member functions of this class are *not* collective.

Parameters

in	*vector*	The vector to pin the memory of.
in	*mode*	The grb::IOMode.

The *mode* argument is *optional*. The default is grb::PARALLEL.

Performance semantics (#IOMode::SEQUENTIAL):

1. This function contains $\Theta(n)$ work, where n is the global length of *vector*.
2. This function moves up to $\mathcal{O}(n)$ bytes of data within its process.
3. This function incurs an inter-process communication cost bounded by $\mathcal{O}(ng + \log(p)l)$.
4. This function may allocate $\mathcal{O}(n)$ memory and (thus) incur system calls.

Performance semantics (#IOMode::PARALLEL):

1. This function contains $\Theta(1)$ work.
2. This function moves $\Theta(1)$ data within its process.
3. This function has no inter-process communication cost.
4. This function performs no dynamic memory allocations and shall not make system calls.

9.56.2.2 PinnedVector() [2/2]

```
PinnedVector ( )  [inline]
```

Base constructor for this class.

This corresponds to pinning an empty vector of zero size in PARALLEL mode. A call to this function inherits the same performance semantics as described above.

Unlike the above, and exceptionally, calling this constructor need not be a collective operation.

9.56.2.3 ~PinnedVector()

```
~PinnedVector ( )  [inline]
```

Destroys a grb::PinnedVector instance.

Destroying a pinned vector will only remove the underlying vector data if and only if: 1) the original grb::Vector has been destroyed; 2) no other PinnedVector instance derived from the same source container exists.

9.56.3 Member Function Documentation

9.56.3.1 getNonzeroIndex()

```
size_t getNonzeroIndex (
            const size_t k ) const  [inline], [noexcept]
```

Retrieves a nonzero index.

Parameters

in	*k*	The nonzero ID to return the index of.

A nonzero is a tuple of an index and nonzero value. A pinned vector holds nonzeroes nonzeroes. Therefore, *k* must be less than nonzeroes.

Returns

The requested index.

Performance semantics.

1. This function incurs $\Theta(1)$ work.
2. This function moves $\Theta(1)$ bytes of data.
3. This function does not incur inter-process communication.
4. This function does not allocate new memory nor makes any other system calls.

9.56.3.2 getNonzeroValue() [1/2]

```
IOType getNonzeroValue (
            const size_t k ) const  [inline], [noexcept]
```

Direct access variation of the general getNonzeroValue function.

This variant is only defined when *IOType* is not `void`.

Warning

If, in your application, *IOType* is templated and can be `void`, then robust code should use the general getNonzeroValue variant.

For semantics, including performance semantics, see the general specification of getNonzeroValue.

Note

By providing this variant, implementations may avoid the requirement that *IOType* must be default-constructable.

9.56.3.3 getNonzeroValue() [2/2]

```
OutputType getNonzeroValue (
            const size_t k,
            const OutputType one = OutputType() ) const  [inline], [noexcept]
```

Returns a requested nonzero of the pinned vector.

Template Parameters

OutputType	The value type returned by this function. If this differs from *IOType* and *IOType* is not `void`, then nonzero values will

Warning

If *OutputType* and *IOType* is not compatible, then this function should not be used.

Parameters

`in`	*k*	The nonzero ID to return the value of.
`in`	*one*	(Optional.) In case *IOType* is `void`, which value should be returned in lieu of a vector element value. By default, this will be a default-constructed instance of *OutputType*.

If *OutputType* cannot be default-constructed, then *one* no longer is optional.

A nonzero is a tuple of an index and nonzero value. A pinned vector holds nonzeroes nonzeroes. Therefore, *k* must be less than nonzeroes.

Returns

The requested value.

Performance semantics.

1. This function incurs $\Theta(1)$ work.
2. This function moves $\Theta(1)$ bytes of data.
3. This function does not incur inter-process communication.
4. This function does not allocate new memory nor makes any other system calls.

9.56.3.4 nonzeroes()

```
size_t nonzeroes ( ) const  [inline], [noexcept]
```

Returns

The number of nonzeroes this pinned vector contains.

Performance semantics.

1. This function incurs $\Theta(1)$ work.
2. This function moves $\Theta(1)$ bytes of data.
3. This function does not incur inter-process communication.
4. This function does not allocate new memory nor makes any other system calls.

9.56.3.5 size()

```
size_t size ( ) const  [inline], [noexcept]
```

Returns

The length of this vector, in number of elements.

Performance semantics.

1. This function incurs $\Theta(1)$ work.
2. This function moves $\Theta(1)$ bytes of data.
3. This function does not incur inter-process communication.
4. This function does not allocate new memory nor makes any other system calls.

The documentation for this class was generated from the following file:

- pinnedvector.hpp

9.57 PIPELINE Class Reference

Configuration parameters relating to the pipeline data structure.

```
#include <config.hpp>
```

Static Public Attributes

- static constexpr const size_t max_containers = 16

  Pipelines are constructed with default space for this many containers.
- static constexpr const size_t max_depth = 16

  Pipelines are constructed with default space for this many stages.
- static constexpr const size_t max_pipelines = 4

  How many independent pipelines any ALP algorithm may concurrently expose.
- static constexpr const size_t max_tiles = 1 << 16

  Pipelines are constructed with default space for this many tiles.
- static constexpr const bool **warn_if_exceeded** = true

  Emit a warning to standard error stream if the default pipeline capacities are exceeded.
- static constexpr const bool **warn_if_not_native** = true

  When `true`, calling a fake nonblocking primitive for a first time will emit a warning to the standard error stream.

9.57.1 Detailed Description

Configuration parameters relating to the pipeline data structure.

9.57.2 Member Data Documentation

9.57.2.1 max_containers

```
constexpr const size_t max_containers = 16  [static], [constexpr]
```

Pipelines are constructed with default space for this many containers.

The default is such that each underlying set used by the pipeline representation takes less than one kB space.

Pipelines could exceed this maximum number of containers. If this happens, and if grb::config::PIPELINE::warn_if_exceeded is configured `true`, a warning will be output to the standard error stream.

9.57.2.2 max_depth

```
constexpr const size_t max_depth = 16  [static], [constexpr]
```

Pipelines are constructed with default space for this many stages.

Pipelines could exceed this number of stages. If this happens, and if grb::config::PIPELINE::warn_if_exceeded is configured `true`, a warning will be output to the standard error stream.

9.57.2.3 max_pipelines

```
constexpr const size_t max_pipelines = 4  [static], [constexpr]
```

How many independent pipelines any ALP algorithm may concurrently expose.

The number of pipelines could exceed this maximum number. If this happens, and if grb::config::PIPELINE::warn_if_exceeded is configured `true`, a warning will be output to the standard error stream.

9.57.2.4 max_tiles

```
constexpr const size_t max_tiles = 1 << 16  [static], [constexpr]
```

Pipelines are constructed with default space for this many tiles.

Pipelines could exceed this number of tiles. If this happens, and if

grb::config::PIPELINE::warn_if_exceeded is configured `true`, a warning will be output to the standard error stream.

The documentation for this class was generated from the following file:

- nonblocking/config.hpp

9.58 PREFETCHING< backend > Class Template Reference

Default prefetching settings for reference and reference_omp backends.

```
#include <config.hpp>
```

Static Public Member Functions

- static constexpr size_t distance ()

  The prefetch distance used during level-2 and level-3 operations.
- static constexpr bool **enabled** ()

  Whether prefetching is enabled.

9.58.1 Detailed Description

template<Backend backend>
class grb::config::PREFETCHING< backend >

Default prefetching settings for reference and reference_omp backends.

Note

By default, prefetching is turned OFF as we found no setting that will never result in a performance degradation across the dataset, workloads, and architectures in our standard test set.

The defaults may be overridden by specialisation, which additionally makes it possible to choose different distances for different backends.

Prefetching presently only is implemented and evaluated for the SpMV and the SpMSpV multiplication kernels. Furthermore, it is only implemented for the gathering variant of either kernel. If you wish further support or evaluation, please feel free to create an issue or to contribute a merge request.

9.58.2 Member Function Documentation

9.58.2.1 distance()

```
static constexpr size_t distance ( )  [inline], [static], [constexpr]
```

The prefetch distance used during level-2 and level-3 operations.

This value will be ignored if enabled() returns `false`.

The documentation for this class was generated from the following file:

- reference/config.hpp

9.59 Pregel< MatrixEntryType > Class Template Reference

A Pregel run-time instance.

```
#include <pregel.hpp>
```

Public Member Functions

- template<typename IType >
  Pregel (const size_t _m, const size_t _n, IType _start, const IType _end, const grb::IOMode _mode)
  Constructs a Pregel instance from input iterators over some graph.
- template<class Op , template< typename > class Id, class Program , typename IOType , typename GlobalProgramData , typename IncomingMessageType , typename OutgoingMessageType >
  grb::RC execute (const Program program, grb::Vector< IOType > &vertex_state, const GlobalProgramData &data, grb::Vector< IncomingMessageType > &in, grb::Vector< OutgoingMessageType > &out, size_t &rounds, grb::Vector< OutgoingMessageType > &out_buffer=grb::Vector< OutgoingMessageType >(0), const size_t max_rounds=0)
  Executes a given vertex-centric program on this graph.
- const grb::Matrix< MatrixEntryType > & get_matrix () const noexcept
  Returns the ALP/GraphBLAS matrix representation of the underlying graph.
- size_t num_edges () const noexcept
  Queries the number of edges of the graph this Pregel instance has been constructed over.
- size_t num_vertices () const noexcept
  Queries the maximum vertex ID for programs running on this Pregel instance.

9.59.1 Detailed Description

template<typename MatrixEntryType>
class grb::interfaces::Pregel< MatrixEntryType >

A Pregel run-time instance.

Pregel wraps around graph data and executes computations on said graph. A runtime thus is constructed from graph, and enables running any Pregel algorithm on said graph.

9.59.2 Constructor & Destructor Documentation

9.59.2.1 Pregel()

```
Pregel (
            const size_t _m,
            const size_t _n,
            IType _start,
            const IType _end,
            const grb::IOMode _mode )  [inline]
```

Constructs a Pregel instance from input iterators over some graph.

Template Parameters

IType	The type of the input iterator.

Parameters

in	*_m*	The maximum vertex ID for excident edges.
in	*_n*	The maximum vertex ID for incident edges.

Note

This is equivalent to the row- and column- size of an input matrix which represents the input graph.

If these values are not known, please scan the input iterators to derive these values prior to calling this constructor. On compelling reasons why such functionality would be useful to provide as a standard factory method, please feel welcome to submit an issue.

Warning

The graph is assumed to have contiguous IDs – i.e., every vertex ID in the range of 0 (inclusive) to the maximum of *m* and *n* (exclusive) has at least one excident or at least one incident edge.

Parameters

in	*_start*	An iterator pointing to the start element of an a collection of edges.
in	*_end*	An iterator matching *_start* in end position.

All edges to be ingested thus are contained within *_start* and *end*.

Parameters

in	*_mode*	Whether sequential or parallel I/O is to be used.

The value of *_mode* only takes effect when there are multiple user processes, such as for example when executing over a distributed-memory cluster. The choice between sequential and parallel I/O should be thus:

- If the edges pointed to by *_start* and *_end* correspond to the *entire* set of edges on *each* process, then the I/O mode should be grb::SEQUENTIAL;
- If the edges pointed to by *_start* and *_end* correspond to *different* sets of edges on each different process while their union represents the graph to be ingested, then the I/O mode should be grb::PARALLEL.

On errors during ingestion, this constructor throws exceptions.

9.59.3 Member Function Documentation

9.59.3.1 execute()

```
grb::RC execute (
            const Program program,
            grb::Vector< IOType > & vertex_state,
            const GlobalProgramData & data,
            grb::Vector< IncomingMessageType > & in,
            grb::Vector< OutgoingMessageType > & out,
            size_t & rounds,
            grb::Vector< OutgoingMessageType > & out_buffer = grb::Vector< OutgoingMessageType >(0),
            const size_t max_rounds = 0 )  [inline]
```

Executes a given vertex-centric *program* on this graph.

The program must be a static function that returns void and takes five input arguments:

- a reference to a vertex-defined state. The type of this reference may be defined by the program, but has to match the element type of *vertex_state* passed to this function.
- a const-reference to an incoming message. The type of this reference may be defined by the program, but has to match the element type of *in* passed to this function. It must furthermore be compatible with the domains of *Op* (see below).
- a reference to an outgoing message. The type of this reference may be defined by the program, but has to match the element type of *out* passed to this function. It must furthermore be compatible with the domains of *Op* (see below).
- a const-reference to a program-defined type. The function of this argument is to collect global read-only algorithm parameters.
- a reference to an instance of grb::interfaces::PregelState. The function of this argument is two-fold: 1) make available global read- only statistics of the graph the algorithm is executing on, and to 2) control algorithm termination conditions.

The program will be called during each round of a Pregel computation. The program is expected to compute something based on the incoming message and vertex-local state, and (optionally) generate an outgoing message. After each round, the outgoing message at all vertices are broadcast to all its neighbours. The Pregel runtime, again for each vertex, reduces all incoming messages into a single message, after which the next round of computation starts, after which the procedure is repeated.

The program terminates in one of two ways:

1. there are no more active vertices; or
2. all active vertices vote to halt.

On program start, i.e., during the first round, all vertices are active. During the computation phase, any vertex can set itself inactive for subsequent rounds by setting grb::interfaces::PregelState::active to `false`. Similarly, any active vertex can vote to halt by setting grb::interfaces::PregelState::voteToHalt to `true`.

Reduction of incoming messages to a vertex will occur through an user- defined monoid given by:

Template Parameters

Op	The binary operation of the monoid. This includes its domain.
Id	The identity element of the monoid.

The following template arguments will be automatically inferred:

Template Parameters

Program	The type of the program to-be executed.
IOType	The type of the state of a single vertex.
GlobalProgramData	The type of globally accessible read-only program data.
IncomingMessageType	The type of an incoming message.
OutgoingMessageType	The type of an outgoing message.

The arguments to this function are as follows:

Parameters

`in`	*program*	The vertex-centric program to execute.

The same Pregel runtime instance hence can be re-used to execute multiple algorithms on the same graph.

Vertex-centric programs have both vertex-local and global state:

Parameters

`in`	*vertex_state*	A vector that contains the state of each vertex.
`in`	*data*	Global read-only state for the given *program*.

The capacity, size, and number of nonzeroes of *vertex_state* must equal the maximum vertex ID.

Finally, in the ALP spirit which aims to control all relevant performance aspects, the workspace required by the Pregel runtime must be pre- allocated and passed in:

Parameters

in	*in*	Where incoming messages are stored. Any initial values may or may not be ignored, depending on the *program* behaviour during the first round of computation.
in	*out*	Where outgoing messages are stored. Any initial values will be ignored.

The capacities and sizes of *in* and *out* must equal the maximum vertex ID. For sparse vectors *in* with more than zero nonzeroes, all initial contents will be overwritten by the identity of the reduction monoid. Any initial contents for *out* will always be ignored as every round of computation starts with the outgoing message set to the monoid identity.

Note

Thus if the program requires some initial incoming messages to be present during the first round of computation, those may be passed as part of a dense vectors *in*.

The contents of *in* and *out* after termination of a vertex-centric function are undefined, including when this function returns grb::SUCCESS. Output of the program should be part of the vertex-centric state recorded in *vertex_state*.

Some statistics are returned after a vertex-centric program terminates:

Parameters

out	*rounds*	The number of rounds the Pregel program has executed. The initial value to *rounds* will be ignored.

The contents of this field shall be undefined when this function does not return grb::SUCCESS.

Vertex-programs execute in rounds and could, if the given program does not infer proper termination conditions, run forever. To curb the number of rounds, the following *optional* parameter may be given:

Parameters

in	*out_buffer*	An optional buffer area that should only be set whenever the config::out_sparsify configuration parameter is not set to config::NONE. If that is the case, then *out_buffer* should have size and capacity equal to the maximum vertex ID.

Parameters

in	*max_rounds*	The maximum number of rounds the *program* may execute. Once reached and not terminated, the program will forcibly terminate.

To turn off termination after a maximum number of rounds, *max_rounds* may be set to zero. This is also the default.

Executing a Pregel function returns one of the following error codes:

Returns

grb::SUCCESS The *program* executed (and terminated) successfully.

grb::MISMATCH At least one of *vertex_state*, *in*, or *out* is not of the required size.

grb::ILLEGAL At least one of *vertex_state*, *in*, or *out* does not have the required capacity.

grb::ILLEGAL If *vertex_state* is not dense.

grb::PANIC In case an unrecoverable error was encountered during execution.

9.59.3.2 get_matrix()

```
const grb::Matrix< MatrixEntryType > & get_matrix ( ) const  [inline], [noexcept]
```

Returns the ALP/GraphBLAS matrix representation of the underlying graph.

This is useful when an application prefers to sometimes use vertex- centric algorithms and other times prefers direct ALP/GraphBLAS algorithms.

Returns

The underlying ALP/GraphBLAS matrix corresponding to the underlying graph.

9.59.3.3 num_edges()

```
size_t num_edges ( ) const  [inline], [noexcept]
```

Queries the number of edges of the graph this Pregel instance has been constructed over.

Returns

The number of edges within the underlying graph.

9.59.3.4 num_vertices()

```
size_t num_vertices ( ) const  [inline], [noexcept]
```

Queries the maximum vertex ID for programs running on this Pregel instance.

Returns

The maximum vertex ID.

The documentation for this class was generated from the following file:

- pregel.hpp

9.60 PregelState Struct Reference

The state of the vertex-center Pregel program that the user may interface with.

```
#include <pregel.hpp>
```

Public Attributes

- bool & active
  Represents whether the current vertex is active.
- const size_t & **indegree**
  The in-degree of this vertex.
- const size_t & **num_edges**
  The number of edges in the global graph.
- const size_t & **num_vertices**
  The number of vertices in the global graph.
- const size_t & **outdegree**
  The out-degree of this vertex.
- const size_t & **round**
  The current round the vertex-centric program is currently executing.
- const size_t & vertexID
  A unique ID of this vertex.
- bool & voteToHalt
  Represents whether this (active) vertex votes to terminate the program.

9.60.1 Detailed Description

The state of the vertex-center Pregel program that the user may interface with.

The state includes global data as well as vertex-centric state. The global state is umodifiable and includes:

- grb::interfaces::PregelState::num_vertices,
- grb::interfaces::PregelState::num_edges, and
- grb::interfaces::PregelState::round.

Vertex-centric state can be either constant or modiable:

- static vertex-centric state: grb::interfaces::PregelState::indegree, grb::interfaces::PregelState::outdegree, and grb::interfaces::PregelState::vertexID.
- modifiable vertex-centric state: grb::interfaces::PregelState::voteToHalt, and grb::interfaces::PregelState::active.

9.60.2 Member Data Documentation

9.60.2.1 active

```
bool& active
```

Represents whether the current vertex is active.

Since this struct is only to-be used within the computational phase of a vertex-centric program, this always reads `true` on the start of a round.

The program may set this field to `false` which will cause this vertex to no longer trigger computational steps during subsequent rounds.

An inactive vertex will no longer broadcast messages.

If all vertices are inactive the program terminates.

9.60.2.2 vertexID

```
const size_t& vertexID
```

A unique ID of this vertex.

This number is an unsigned integer between 0 (inclusive) and the number of vertices the underlying graph holds (exclusive).

9.60.2.3 voteToHalt

```
bool& voteToHalt
```

Represents whether this (active) vertex votes to terminate the program.

On start of a round, this entry is set to `false`. If all active vertices set this to `true`, the program will terminate after the current round.

The documentation for this struct was generated from the following file:

- pregel.hpp

9.61 Properties< backend > Class Template Reference

Collection of various properties on the given ALP/GraphBLAS *backend*.

```
#include <properties.hpp>
```

Static Public Attributes

- static constexpr const bool isBlockingExecution = true

  Whether the given backend supports blocking execution or is, instead, non-blocking.
- static constexpr const bool isNonblockingExecution = !isBlockingExecution

  Whether the given backend is non-blocking or is, instead, blocking.
- static constexpr const bool writableCaptured = true

  Whether a scalar, non-ALP/GraphBLAS object, may be captured by and written to by a lambda function that is passed to grb::eWiseLambda.

9.61.1 Detailed Description

template<enum Backend backend>
class grb::Properties< backend >

Collection of various properties on the given ALP/GraphBLAS *backend*.

Template Parameters

backend	The backend of which to access its properties.

The properties collected here are meant to be compile-time constants that provide insight in what features the given *backend* supports. ALP user code may rely on the properties specified herein. All ALP backends must define all properties here specified.

The default template class shall be empty in order to ensure implementing backends must specialise this class, while also making sure no backend may accidentally implicitly and erroneously propagate global defaults.

9.61.2 Member Data Documentation

9.61.2.1 isBlockingExecution

```
constexpr const bool isBlockingExecution = true  [static], [constexpr]
```

Whether the given *backend* supports blocking execution or is, instead, non-blocking.

In blocking execution mode, any ALP/GraphBLAS primitive, when it returns, is guaranteed to have completed the requested computation.

If a given *backend* has this property `true` then the isNonblockingExecution property must read `false`, and vice versa.

9.61.2.2 isNonblockingExecution

```
constexpr const bool isNonblockingExecution = !isBlockingExecution  [static], [constexpr]
```

Whether the given *backend* is non-blocking or is, instead, blocking.

In non-blocking execution mode, any ALP/GraphBLAS primitive, on return, *may* in fact *not* have completed the requested computation.

Non-blocking execution thus allows for the lazy evaluation of an ALP code, which, in turn, allows for cross-primitive optimisations to be automatically applied.

If a given *backend* has this property `true` then the isBlockingExecution property must read `false`, and vice versa.

9.61.2.3 writableCaptured

```
constexpr const bool writableCaptured = true  [static], [constexpr]
```

Whether a scalar, non-ALP/GraphBLAS object, may be captured by and written to by a lambda function that is passed to grb::eWiseLambda.

Typically, if the *backend* is shared-memory parallel, this function would return `false`. Purely Single Program, Multiple Data (SPMD) backends over distributed memory, including simple sequential backends, would have this property return `true`.

Notably, hybrid SPMD + OpenMP backends (e.g., grb::hybrid), are not pure SPMD and as such would return `false`.

See also

grb::eWiseLambda()

The documentation for this class was generated from the following file:

- properties.hpp

9.62 relu< D1, D2, D3, implementation > Class Template Reference

This operation is equivalent to grb::operators::min.

`#include <ops.hpp>`

Inherits Operator< internal::relu< D1, D1, D1, config::default_backend > >.

9.62.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::relu< D1, D2, D3, implementation >

This operation is equivalent to grb::operators::min.

It assumes that the right-hand input is the bias, while the left-hand input is the signal.

See also

min

The documentation for this class was generated from the following file:

- ops.hpp

9.63 right_assign< D1, D2, D3, implementation > Class Template Reference

This operator discards all left-hand side input and simply copies the right-hand side input to the output variable.

`#include <ops.hpp>`

Inherits Operator< internal::right_assign< D1, D1, D1, config::default_backend > >.

9.63.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::right_assign< D1, D2, D3, implementation >

This operator discards all left-hand side input and simply copies the right-hand side input to the output variable.

It exposes the complete interface detailed in grb::operators::internal::Operator. This operator can be passed to any GraphBLAS function or object constructor.

Mathematical notation: $\odot(x, y) \rightarrow y$.

Template Parameters

D1	The left-hand side input domain.
D2	The right-hand side input domain.
D3	The output domain.

The documentation for this class was generated from the following file:

- ops.hpp

9.64 right_assign_if< D1, D2, D3, implementation > Class Template Reference

This operator assigns the right-hand input if the left-hand input evaluates `true`.

`#include <ops.hpp>`

Inherits Operator< internal::right_assign_if< D1, D1, D1, config::default_backend > >.

9.64.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::right_assign_if< D1, D2, D3, implementation >

This operator assigns the right-hand input if the left-hand input evaluates `true`.

If the left-hand input does not evaluate `true`, then the output field is unmodified.

Warning

Therefore, this operator may propagate the use of uninitialised values if not used with care. Ensuring its use with in-place primitives is recommended.

The documentation for this class was generated from the following file:

- ops.hpp

9.65 Semiring< _OP1, _OP2, _ID1, _ID2 > Class Template Reference

A generalised semiring.

`#include <semiring.hpp>`

Public Types

- typedef Monoid< _OP1, _ID1 > **AdditiveMonoid**

  The additive monoid type.
- typedef _OP1 **AdditiveOperator**

  The additive operator type.
- typedef _OP2::D1 **D1**

  The first input domain of the multiplicative operator.
- typedef _OP2::D2 **D2**

  The second input domain of the multiplicative operator.
- typedef _OP2::D3 D3

  The output domain of the multiplicative operator.
- typedef _OP1::D2 D4

  The second input domain of the additive operator.
- typedef Monoid< _OP2, _ID2 > **MultiplicativeMonoid**

  The multiplicative monoid type.
- typedef _OP2 **MultiplicativeOperator**

  The multiplicative operator type.
- template<typename OneType >
  using **One** = _ID2< OneType >

  The identity under multiplication.
- template<typename ZeroType >
  using **Zero** = _ID1< ZeroType >

  The identity under addition.

Public Member Functions

- AdditiveMonoid getAdditiveMonoid () const

  Retrieves the underlying additive monoid.
- AdditiveOperator getAdditiveOperator () const

  Retrieves the underlying additive operator.
- MultiplicativeMonoid getMultiplicativeMonoid () const

  Retrieves the underlying multiplicative monoid.
- MultiplicativeOperator getMultiplicativeOperator () const

  Retrieves the underlying multiplicative operator.
- template<typename D >
  constexpr D getOne () const

  Sets the given value equal to one, corresponding to this semiring.
- template<typename D >
  constexpr D getZero () const

  Retrieves the zero corresponding to this semiring.

Static Public Attributes

- static constexpr size_t **blocksize** = blocksize_mul < blocksize_add ? blocksize_mul : blocksize_add

  Blocksize for element-wise multiply-adds.
- static constexpr size_t **blocksize_add** = D3_bsz < D4_bsz ? D3_bsz : D4_bsz

  Blocksize for element-wise addition.
- static constexpr size_t **blocksize_mul** = mul_input_bsz < D3_bsz ? mul_input_bsz : D3_bsz

  Blocksize for element-wise multiplication.

9.65.1 Detailed Description

template<class _OP1, class _OP2, template< typename > class _ID1, template< typename > class _ID2>
class grb::Semiring< _OP1, _OP2, _ID1, _ID2 >

A generalised semiring.

This semiring works with the standard operators provided in grb::operators as well as with standard identities provided in grb::identities.

Operators

An operator *OP* here is of the form $f : D_1 \times D_2 \to D_3$; i.e., it has a fixed left-hand input domain, a fixed right-hand input domain, and a fixed output domain.

A generalised semiring must include two operators; an additive operator, and a multiplicative one:

1. $\oplus : D_1 \times D_2 \to D_3$, and
2. $\otimes : D_4 \times D_5 \to D_6$.

By convention, primitives such as grb::mxv will feed the output of the multiplicative operation to the additive operator as left-hand side input; hence, a valid semiring must have $D_6 = D_1$. Should the additive operator reduce several multiplicative outputs, the thus-far accumulated value will thus be passed as right-hand input to the additive operator; hence, a valid semiring must also have $D_2 = D_3$.

If these constraints on the domains do not hold, attempted compilation will result in a clear error message.

A semiring, in our definition here, thus in fact only defines four domains. We may thus rewrite the above definitions of the additive and multiplicative operators as:

1. $\otimes : D_1 \times D_2 \to D_3$, and
2. $\oplus : D_3 \times D_4 \to D_4$.

Identities

There are two identities that make up a generalised semiring: the zero- identity and the one-identity. These identities must be able to instantiate values for different domains, should indeed the four domains a generalised semiring operates on differ.

Specifically, the zero-identity may be required for any of the domains the additive and multiplicative operators employ, whereas the one-identity may only be required for the domains the multiplicative operator employs.

Standard examples

An example of the standard semiring would be: grb::Semiring< grb::operators::add< double, double, double >, grb::operators::mul< double, double, double >, grb::identities::zero, grb::identitites::one

realSemiring;

In this standard case, all domains the operators the semiring comprises are equal to one another. GraphBLAS supports the following shorthand for this special case: grb::Semiring< grb::operators::add< double >, grb::operators↩ ::mul< double >, grb::identities::zero, grb::identities::one

realSemiring;

As another example, consider min-plus algebras. These may be used, for example, for deriving shortest paths through an edge-weighted graph: grb::Semiring< grb::operators::min< unsigned int >, grb::operators::add< unsigned int >, grb::identities::negative_infinity, grb::identities::zero

minPlus;

CMonoid-categories

While in these standard examples the relation to standard semirings as defined in mathematics apply, the possiblity of having differing domains that may not even be subsets of one another makes the above sketch generalisation incompatible with the standard notion of semirings.

Our notion of a generalised semiring indeed is closer to what one might call CMonoid-categories, i.e. categories enriched in commutative monoids. Such CMonoid-categories are specified by some data, and are required to satisfy certain algebraic (equational) laws, thus being well-specified mathematical objects.

Additionally, such CMonoid-categories encapsulate the definition of semirings, vector spaces, left modules and right modules.

The full structure of a CMonoid-category C is specified by the data:

1. a set ob(C) of so-called objects,
2. for each pair of objects a,b in ob(C), a commutative monoid (C(a,b), 0_{a,b}, +_{a,b}),
3. for each triple of objects a,b,c in ob(C), a multiplication operation ._{a,b,c} : C(b,c) x C(a,b) -> C(a,c), and
4. for each object a in ob(C), a multiplicative identity 1_a in C(a,a).

This data is then required to specify a list of algebraic laws that essentially capture:

1. (that the (C(a,b), 0_{a,b}, +_{a,b}) are commutative monoids)
2. joint associativity of the family of multiplication operators,
3. that the multiplicative identities 1_a are multiplicative identities,
4. that the family of multiplication operators ._{a,b,c} distributes over the family of addition operators +_{a,b} on the left and on the right in an appropriate sense, and
5. left and right annihilativity of the family of additive zeros 0_{a,b}.

Generalised semirings in terms of CMonoid-categories

The current notion of generalised semiring is specified by the following data:

1. operators OP1, OP2,
2. the four domains those operators are defined on,
3. an additive identity ID1, and
4. a multiplicative identity ID2.

The four domains correspond to the choice of a CMonoid-category with two objects; e.g., $ob(C) = \{a, b\}$. This gives rise to four possible pairings of the objects, including self-pairs, that correspond to the four different domains.

CMonoid-categories then demand an additive operator must exist that operates purely within each of the four domains, when combined with a zero identity that likewise must exist in each of the four domains. None of these additive operators in fact matches with the generalised semiring's additive operator.

CMonoid-categories also demand the existance of six different multiplicative operators that operate on three different domains each, that the composition of these operators is associative, that these operators distribute over the appropriate additive operators, and that there exists an multiplicative identity over at least one of the input domains.

One of these six multiplicative operators is what appears in our generalised semiring. We seem to select exactly that multiplicative operator for which both input domains have an multiplicative identity.

Finally, the identities corresponding to additive operators must act as annihilators over the matching multiplicative operators.

Full details can be found in the git repository located here: https://gitlab.huaweirc.ch/abooij/semirings

Template Parameters

_OP1	The addition operator.
_OP2	The multiplication operator.
_ID1	The identity under addition (the '0').
_ID2	The identity under multiplication (the '1').

9.65.2 Member Typedef Documentation

9.65.2.1 D3

```
typedef _OP2::D3 D3
```

The output domain of the multiplicative operator.

The first input domain of the additive operator.

9.65.2.2 D4

```
typedef _OP1::D2 D4
```

The second input domain of the additive operator.

The output domain of the additive operator.

9.65.3 Member Function Documentation

9.65.3.1 getAdditiveMonoid()

```
AdditiveMonoid getAdditiveMonoid ( ) const  [inline]
```

Retrieves the underlying additive monoid.

Returns

The underlying monoid. Any state is copied.

9.65.3.2 getAdditiveOperator()

```
AdditiveOperator getAdditiveOperator ( ) const  [inline]
```

Retrieves the underlying additive operator.

Returns

The underlying operator. Any state is copied.

9.65.3.3 getMultiplicativeMonoid()

```
MultiplicativeMonoid getMultiplicativeMonoid ( ) const  [inline]
```

Retrieves the underlying multiplicative monoid.

Returns

The underlying monoid. Any state is copied.

9.65.3.4 getMultiplicativeOperator()

```
MultiplicativeOperator getMultiplicativeOperator ( ) const  [inline]
```

Retrieves the underlying multiplicative operator.

Returns

The underlying operator. Any state is copied.

9.65.3.5 getOne()

```
constexpr D getOne ( ) const  [inline], [constexpr]
```

Sets the given value equal to one, corresponding to this semiring.

The identity value will be cast to the requested domain.

Template Parameters

D	The requested domain of the one. The arbitrary choice for the default return type is *D1*-- the reasoning being to simply have th

Returns

The one corresponding to this semiring, cast to the requested domain.

9.65.3.6 getZero()

```
constexpr D getZero ( ) const  [inline], [constexpr]
```

Retrieves the zero corresponding to this semiring.

The zero value will be cast to the requested domain.

Template Parameters

D	The requested domain of the zero. The arbitrary choice for the default return type is *D1*-- inspired by the regularly occurring ex

Returns

The zero corresponding to this semiring, cast to the requested domain.

The documentation for this class was generated from the following file:

- semiring.hpp

9.66 SIMD_SIZE Class Reference

The SIMD size, in bytes.

`#include <config.hpp>`

9.66.1 Detailed Description

The SIMD size, in bytes.

The documentation for this class was generated from the following file:

- base/config.hpp

9.67 spmd< implementation > Class Template Reference

For backends that support multiple user processes this class defines some basic primitives to support SPMD programming.

`#include <spmd.hpp>`

Static Public Member Functions

- static size_t nprocs () noexcept
- static size_t pid () noexcept

9.67.1 Detailed Description

template<Backend implementation>
class grb::spmd< implementation >

For backends that support multiple user processes this class defines some basic primitives to support SPMD programming.

All backends must implement this interface, including backends that do not support multiple user processes. The interface herein defined hence ensures to allow for trivial implementations for single user process backends.

9.67.2 Member Function Documentation

9.67.2.1 nprocs()

```
static size_t nprocs ( )  [inline], [static], [noexcept]
```

Returns

The number of user processes in this ALP run.

9.67.2.2 pid()

```
static size_t pid ( )  [inline], [static], [noexcept]
```

Returns

The ID of this user process.

The documentation for this class was generated from the following file:

- spmd.hpp

9.68 square_diff< D1, D2, D3, implementation > Class Template Reference

This operation returns the squared difference between two numbers.

`#include <ops.hpp>`

Inherits Operator< internal::square_diff< D1, D2, D3, config::default_backend > >.

9.68.1 Detailed Description

template<typename D1, typename D2, typename D3, enum Backend implementation = config::default_backend>
class grb::operators::square_diff< D1, D2, D3, implementation >

This operation returns the squared difference between two numbers.

Mathematical notation: $\odot(x, y) \to (x - y)^2$.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator- and operator* overloads available.

See also

abs_diff

The documentation for this class was generated from the following file:

- ops.hpp

9.69 subtract< D1, D2, D3, implementation > Class Template Reference

Numerical substraction of two numbers.

`#include <ops.hpp>`

Inherits Operator< internal::substract< D1, D1, D1, config::default_backend > >.

9.69.1 Detailed Description

template<typename D1, typename D2 = D1, typename D3 = D2, enum Backend implementation = config::default_backend>
class grb::operators::subtract< D1, D2, D3, implementation >

Numerical substraction of two numbers.

Mathematical notation: $\odot(x, y) \rightarrow x - y$.

Note

This is the inverse to grb::operators::add.

Warning

This operator expects numerical types for *D1*, *D2*, and *D3*, or types that have the appropriate operator- overloads available.

The documentation for this class was generated from the following file:

- ops.hpp

9.70 Vector< D, implementation, C > Class Template Reference

A GraphBLAS vector.

`#include <vector.hpp>`

Classes

- class const_iterator
  A standard iterator for the Vector< D > class.

Public Types

- typedef D & lambda_reference
  Defines a reference to a value of type D.
- typedef D **value_type**
  The type of elements stored in this vector.

Public Member Functions

- Vector (const size_t n)

  Creates an ALP/GraphBLAS vector.
- Vector (const size_t n, const size_t nz)

  Creates an ALP/GraphBLAS vector.
- Vector (Vector< D, implementation, C > &&x) noexcept

  Move constructor.
- ~Vector ()

  Default destructor.
- const_iterator begin () const

  Same as cbegin().
- template<Descriptor descr = descriptors::no_operation, class Accum = typename operators::right_assign< D, D, D >, typename fwd_iterator = const D * __restrict__>
  RC build (const Accum &accum, const fwd_iterator start, const fwd_iterator end, fwd_iterator npos)

  Copy from raw user-supplied data into a vector.
- template<Descriptor descr = descriptors::no_operation, class Accum = operators::right_assign< D, D, D >, typename ind_iterator = const size_t * __restrict__, typename nnz_iterator = const D * __restrict__, class Dup = operators::right_assign< D, D, D >>
  RC build (const Accum &accum, const ind_iterator ind_start, const ind_iterator ind_end, const nnz_iterator nnz_start, const nnz_iterator nnz_end, const Dup &dup=Dup())

  Copy from raw user-supplied data into a vector.
- template<Descriptor descr = descriptors::no_operation, typename mask_type , class Accum , typename ind_iterator = const size_t * __restrict__, typename nnz_iterator = const D * __restrict__, class Dup = operators::right_assign< D, typename nnz_iterator::value_type, D >>
  RC build (const Vector< mask_type, implementation, C > &mask, const Accum &accum, const ind_iterator ind_start, const ind_iterator ind_end, const nnz_iterator nnz_start, const nnz_iterator nnz_end, const Dup &dup=Dup())

  Copy from raw user-supplied data into a vector.
- const_iterator cbegin () const

  Provides the only mechanism to extract data from this GraphBLAS vector.
- const_iterator cend () const

  Indicates the end to the elements in this container.
- const_iterator end () const

  Same as cend().
- template<typename T >
  RC nnz (T &nnz) const

  Return the number of nonzeroes in this vector.
- template<class Monoid >
  lambda_reference operator() (const size_t i, const Monoid &monoid=Monoid())

  Returns a lambda reference to an element of this sparse vector.
- Vector< D, implementation, C > & operator= (Vector< D, implementation, C > &&x) noexcept

  Move-from-temporary assignment.
- lambda_reference operator[ ] (const size_t i)

  Returns a lambda reference to an element of this vector.
- template<typename T >
  RC size (T &size) const

  Return the dimension of this vector.

9.70.1 Detailed Description

template<typename D, enum Backend implementation, typename C>
class grb::Vector< D, implementation, C >

A GraphBLAS vector.

This is an opaque data type that can be provided to any GraphBLAS function, such as, grb::eWiseMulAdd, for example.

Template Parameters

D	The type of an element of this vector. *D* shall not be a GraphBLAS type.
implementation	Allows different backends to implement different versions of this data type.
C	The type of the class that keeps track of sparsity structure.

Warning

Creating a grb::Vector of other GraphBLAS types is *not allowed*. Passing a GraphBLAS type as template parameter will lead to undefined behaviour.

Note

The implementation found in the same file as this documentation catches invalid backends only. This class should never compile.

See also

grb::Vector< D, reference, C > for an actual implementation example.

9.70.2 Member Typedef Documentation

9.70.2.1 lambda_reference

```
typedef D& lambda_reference
```

Defines a reference to a value of type *D*.

This reference is only valid when used inside a lambda function that is passed to grb::eWiseLambda().

Warning

Any other use of this reference incurs undefined behaviour.

Example.

An example valid use:

```
void f(
     Vector< D >::lambda_reference x,
     const Vector< D >::lambda_reference y,
     const Vector< D > &v
) {
     grb::eWiseLambda( [x,y](const size_t i) {
         x += y;
     }, v );
}
```

This code adds *y* to *x* for every element in *v*. For a more useful example, see grb::eWiseLambda.

Warning

Note that, unlike the above, this below code is illegal since it does not evaluate via a lambda passed to any of the above GraphBLAS lambda functions (such as grb::eWiseLambda).

```
void f(
     Vector< D >::lambda_reference x,
     const Vector< D >::lambda_reference y
) {
     x += y;
}
```

Also this usage is illegal since it does not rely on any GraphBLAS-approved function listed above:

```
void f(
     Vector< D >::lambda_reference x,
     const Vector< D >::lambda_reference y
) {
     std::functional< void() > f =
         [x,y](const size_t i) {
             x += y;
         };
     f();
}
```

There is no similar concept in the official GraphBLAS specs.

See also

grb::Vector::operator[]()

grb::eWiseLambda

9.70.3 Constructor & Destructor Documentation

9.70.3.1 Vector() [1/3]

```
Vector (
            const size_t n,
            const size_t nz )  [inline]
```

Creates an ALP/GraphBLAS vector.

The given dimension will be fixed throughout the lifetime of this container. After instantiation, the vector will contain no nonzeroes.

Parameters

in	*n*	The dimension of this vector.
in	*nz*	The minimal initial capacity of this vector.

The argument *nz* is *optional*. Its default value is *n*.

Performance semantics

A backend must:

1. define cost in terms of work,
2. define intra-process data movement costs,
3. define inter-process data movement costs,
4. define whether inter-process synchronisations occur,
5. define memory storage requirements and may define this in terms of *n* and/or *nz*, and
6. must define whether system calls may be made, and in particular whether allocation or freeing of dynamic memory occurs or may occur.

Warning

Most backends will require work, intra-process data movement, and system calls for the dynamic allocation of memory areas, all of (at least the complexity of) $\Omega(nz)$. Hence avoid the use of this constructor within performance-critical code sections.

9.70.3.2 Vector() [2/3]

```
Vector (
            const size_t n )  [inline]
```

Creates an ALP/GraphBLAS vector.

This constructor is specified as per the above where *nz* is to taken equal to *n*.

9.70.3.3 Vector() [3/3]

```
Vector (
            Vector< D, implementation, C > && x )  [inline], [noexcept]
```

Move constructor.

This will make the new vector equal the given GraphBLAS vector while destroying the supplied GraphBLAS vector.

This function always succeeds and will not throw exceptions.

Parameters

in	*x*	The GraphBLAS vector to move to this new container.

Performance semantics

1. This constructor completes in $\Theta(1)$ time.
2. This constructor does not allocate new data on the heap.
3. This constructor uses $\mathcal{O}(1)$ more memory than already used by this application at constructor entry.
4. This constructor incurs at most $\mathcal{O}(1)$ bytes of data movement.

9.70.3.4 ~Vector()

```
~Vector ( )  [inline]
```

Default destructor.

Frees all associated memory areas.

Performance semantics

1. This destructor contains $\mathcal{O}(n)$ work, where n is the capacity of this vector.
2. This destructor is only allowed to free memory, not allocate.
3. This destructor uses $\mathcal{O}(1)$ more memory than already used by this application at entry.
4. This destructor shall move at most $\mathcal{O}(n)$ bytes of data.
5. This destructor will make system calls.

Warning

Avoid the use of this destructor within performance critical code sections.

Note

Destruction of this GraphBLAS container is the only way to guarantee that any underlying dynamically allocated memory is freed.

9.70.4 Member Function Documentation

9.70.4.1 begin()

```
const_iterator begin ( ) const  [inline]
```

Same as cbegin().

Since iterators are only supplied as a data extraction mechanism, there is no overloaded version of this function that returns a non-const iterator.

9.70.4.2 build() [1/3]

```
RC build (
            const Accum & accum,
            const fwd_iterator start,
            const fwd_iterator end,
            fwd_iterator npos )  [inline]
```

Copy from raw user-supplied data into a vector.

This is the dense unmasked variant.

Template Parameters

descr	The pre-processing descriptor to use.
fwd_iterator	The type of input iterator. By default, this will be a raw *unaliased* pointer.
Accum	The accumulator type used to merge incoming new elements with existing contents, if any.

Parameters

in	*accum*	The accumulator used to merge incoming new elements with existing content, if any.

Parameters

in	*start*	The iterator to the first element that should be copied into this Graph↩ BLAS vector.
in	*end*	Iterator shifted exactly one past the last element that should be copied into this Graph↩ BLAS vector.
out	*npos*	The last iterator position after exiting this function. In most cases this will equal *end*. This parameter is optional.

The first element from *it* will be copied into the element with index 0 in this vector. The k-th element will be copied into the element with index $k-1$. The iterator *start* will be incremented along with k until it compares equal to *end*,

or until it has been incremented *n* times, where *n* is the dimension of this vector. In the latter case, any remaining values are ignored.

Returns

grb::SUCCESS This function always succeeds.

Note

The default accumulator expects *val* to be of the same type as nonzero elements in this function, and will cause old values to be overwritten by the incoming new values.

Previous contents of the vector are retained. If these are to be cleared first, see clear(). The default accumulator is NOT an alternative since any pre-existing values corresponding to entries in the mask that evaluate to false will be retained.

The parameter *n* can be used to ingest only a subset of a larger data structure pointed to by *start*. At the end of the call, *start* will then not be equal to *end*, but instead point to the first element of the remainder of the larger data structure.

Valid descriptors

grb::descriptors::no_operation, grb::descriptors::no_casting.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *accum* must match the type of *val*, 2) the second domain must match the type *D* of nonzeroes in this vector, and 3) the third domain must match *D*. If one of these is not true, the code shall not compile.

Performance semantics

If the capacity of this container is sufficient to perform the requested operation, then:

1. This function contains $\Theta(n)$ work.
2. This function will take at most $\Theta(1)$ memory beyond the memory already used by the application before the call to this function.
3. This function moves at most $n(2sizeof(D) + sizeof(bool)) + \mathcal{O}(1)$ bytes of data.

Performance exceptions

If the capacity of this container at function entry is insufficient to perform the requested operation, then, in addition to the above:

1. this function allocates $\Theta(n)$ bytes of memory .
2. this function frees $\mathcal{O}(n)$ bytes of memory.
3. this function will make system calls.

Note

An implementation may ensure that at object construction the capacity is maximised. In that case, the above performance exceptions will never come to pass.

See also

grb::buildVector for the ALP/GraphBLAS standard dispatcher to this function.

9.70.4.3 build() [2/3]

```
RC build (
            const Accum & accum,
            const ind_iterator ind_start,
            const ind_iterator ind_end,
            const nnz_iterator nnz_start,
            const nnz_iterator nnz_end,
            const Dup & dup = Dup() )  [inline]
```

Copy from raw user-supplied data into a vector.

This is the sparse non-masked variant.

Template Parameters

descr	The pre-processing descriptor to use.
Accum	The type of the operator used to combine newly input data with existing data, if any.
ind_iterator	The type of index input iterator. By default, this will be a raw *unaliased* pointer to elements of type *size_t*.
nnz_iterator	The type of nonzero input iterator. By default, this will be a raw *unaliased* pointer to elements of type *D*.
Dup	The type of operator used to combine any duplicate input values.

Parameters

in	*accum*	The operator to be used when writing back the result of data that was already in this container prior to calling this function.

Parameters

in	*ind_start*	The iterator to the first index value that should be added to this GraphBLAS vector.
in	*ind_end*	Iterator corresponding to the end position of *ind_start*.
in	*nnz_start*	The iterator to the first nonzero value that should be added to this GraphBLAS vector.
in	*nnz_end*	Iterator corresponding to the end position of *nnz_start*.

Parameters

in	*dup*	The operator to be used when handling multiple nonzero values that are to be mapped to the same index position.

The first element from *nnz_start* will be copied into this vector at the index corresponding to the first element from *ind_start*. Then, both nonzero and index value iterators advance to add the next input element and the process repeats until either of the input iterators reach *nnz_end* or *ind_end*, respectively. If at that point one of the iterators still has remaining elements, then those elements are ignored.

Returns

grb::MISMATCH When attempting to insert a nonzero value at an index position that is larger or equal to the dimension of this vector. When this code is returned, the contents of this container are undefined.

grb::SUCCESS When all elements are successfully assigned.

Note

The default accumulator expects *D* to be of the same type as nonzero elements of this operator, and will cause old values to be overwritten by the incoming new values.

The default *dup* expects *D* to be of the same type as nonzero elements of this operator, and will cause duplicate values to be discarded in favour of the last seen value.

Previous contents of the vector are retained. If these are to be cleared first, see clear(). The default accumulator is NOT an alternative since any pre-existing values corresponding to entries in the mask that evaluate to false will be retained.

Valid descriptors

grb::descriptors::no_operation, grb::descriptors::no_casting, grb::descriptors::no_duplicates.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *accum* must match the type of *D*, 2) the second domain must match nnz_iterator::value_type, and 3) the third domain must *D*. If one of these is not true, the code shall not compile.

Performance semantics.

1. This function contains $\Theta(n)$ work.
2. This function will take at most $\Theta(1)$ memory beyond the memory already used by the application before the call to this function.
3. This function moves at most $n(2sizeof(D) + sizeof(bool)) + \mathcal{O}(1)$ bytes of data.

Performance exceptions

If the capacity of this container at function entry is insufficient to perform the requested operation, then, in addition to the above:

1. this function allocates $\Theta(n)$ bytes of memory .
2. this function frees $\mathcal{O}(n)$ bytes of memory.
3. this function will make system calls.

Note

An implementation may ensure that at object construction the capacity is maximised. In that case, the above performance exceptions will never come to pass.

See also

grb::buildVector for the ALP/GraphBLAS standard dispatcher to this function.

9.70.4.4 build() [3/3]

```
RC build (
            const Vector< mask_type, implementation, C > & mask,
            const Accum & accum,
            const ind_iterator ind_start,
            const ind_iterator ind_end,
            const nnz_iterator nnz_start,
            const nnz_iterator nnz_end,
            const Dup & dup = Dup() )  [inline]
```

Copy from raw user-supplied data into a vector.

This is the sparse masked variant.

Template Parameters

descr	The pre-processing descriptor to use.
mask_type	The value type of the *mask* vector. This type is *not* required to be *bool*.
Accum	The type of the operator used to combine newly input data with existing data, if any.
ind_iterator	The type of index input iterator. By default, this will be a raw *unaliased* pointer to elements of type *size_t*.
nnz_iterator	The type of nonzero input iterator. By default, this will be a raw *unaliased* pointer to elements of type *D*.
Dup	The type of operator used to combine any duplicate input values.

Parameters

in	*mask*	An element is only added to this container if its index i has a nonzero at the same position in *mask* that evaluates true.
in	*accum*	The operator to be used when writing back the result of data that was already in this container prior to calling this function.
in	*ind_start*	The iterator to the first index value that should be added to this Graph↩ BLAS vector.

Parameters

in	*ind_end*	Iterator corresponding to the end position of *ind_start*.
in	*nnz_start*	The iterator to the first nonzero value that should be added to this GraphBLAS vector.
in	*nnz_end*	Iterator corresponding to the end position of *nnz_start*.
in	*dup*	The operator to be used when handling multiple nonzero values that are to be mapped to the same index position.

The first element from *nnz_start* will be copied into this vector at the index corresponding to the first element from *ind_start*. Then, both nonzero and index value iterators advance to add the next input element and the process repeats until either of the input iterators reach *nnz_end* or *ind_end*, respectively. If at that point one of the iterators still has remaining elements, then those elements are ignored.

Returns

grb::MISMATCH When attempting to insert a nonzero value at an index position that is larger or equal to the dimension of this vector. When this code is returned, the contents of this container are undefined.

grb::SUCCESS When all elements are successfully assigned.

Note

The default accumulator expects *D* to be of the same type as nonzero elements of this operator, and will cause old values to be overwritten by the incoming new values.

The default *dup* expects *D* to be of the same type as nonzero elements of this operator, and will cause duplicate values to be discarded in favour of the last seen value.

Previous contents of the vector are retained. If these are to be cleared first, see clear(). The default accumulator is NOT an alternative since any pre-existing values corresponding to entries in the mask that evaluate to false will be retained.

Valid descriptors

grb::descriptors::no_operation, grb::descriptors::no_casting, grb::descriptors::invert_mask, grb::descriptors::no_duplicates.

Note

Invalid descriptors will be ignored.

If grb::descriptors::no_casting is specified, then 1) the first domain of *accum* must match the type of *D*, 2) the second domain must match nnz_iterator::value_type, and 3) the third domain must *D*. If one of these is not true, the code shall not compile.

Performance semantics.

1. This function contains $\Theta(n)$ work.
2. This function will take at most $\Theta(1)$ memory beyond the memory already used by the application before the call to this function.
3. This function moves at most $n(2sizeof(D) + sizeof(bool)) + \mathcal{O}(1)$ bytes of data.

Performance exceptions

If the capacity of this container at function entry is insufficient to perform the requested operation, then, in addition to the above:

1. this function allocates $\Theta(n)$ bytes of memory .
2. this function frees $\mathcal{O}(n)$ bytes of memory.
3. this function will make system calls.

Note

An implementation may ensure that at object construction the capacity is maximised. In that case, the above performance exceptions will never come to pass.

See also

grb::buildVector for the ALP/GraphBLAS standard dispatcher to this function.

9.70.4.5 cbegin()

```
const_iterator cbegin ( ) const  [inline]
```

Provides the only mechanism to extract data from this GraphBLAS vector.

The order in which nonzero elements are returned is undefined.

Returns

An iterator pointing to the first element of this vector, if any; *or* an iterator in end position if this vector contains no nonzeroes.

Note

An 'iterator in end position' compares equal to the const_iterator returned by cend().

Performance semantics

1. This function contains $\mathcal{O}(1)$ work.
2. This function is allowed allocate dynamic memory.
3. This function uses up to $\mathcal{O}(1)$ more memory than already used by this application at entry.
4. This function shall move at most $\mathcal{O}(1)$ bytes of data.
5. This function may make system calls.

Warning

Avoid the use of this function within performance critical code sections.

Note

This function may make use of a const_iterator that is buffered, hence possibly causing its implicitly called constructor to allocate dynamic memory.

9.70.4.6 cend()

```
const_iterator cend ( ) const  [inline]
```

Indicates the end to the elements in this container.

Returns

An iterator at the end position of this container.

Performance semantics

1. This function contains $\mathcal{O}(1)$ work.
2. This function is not allowed allocate dynamic memory.
3. This function uses up to $\mathcal{O}(1)$ more memory than already used by this application at entry.
4. This function shall move at most $\mathcal{O}(1)$ bytes of data.
5. This function shall *not* induce any system calls.

Note

Even if cbegin() returns a buffered const_iterator that may require dynamic memory allocation and additional data movement, this specification disallows the same to happen for the construction of an iterator in end position.

9.70.4.7 end()

```
const_iterator end ( ) const  [inline]
```

Same as cend().

Since iterators are only supplied as a data extraction mechanism, there is no overloaded version of this function that returns a non-const iterator.

9.70.4.8 nnz()

```
RC nnz (
            T & nnz ) const  [inline]
```

Return the number of nonzeroes in this vector.

Template Parameters

T	The integral output type.

Parameters

out	*nnz*	Where to store the number of nonzeroes contained in this vector. Its initial value is ignored.

Returns

grb::SUCCESS When the function call completes successfully.

Note

This function cannot fail.

Performance semantics

This function

1. contains $\Theta(1)$ work,
2. will not allocate new dynamic memory,
3. will take at most $\Theta(1)$ memory beyond the memory already used by the application before the call to this function.
4. will move at most $sizeof(T) + sizeof(size\_t)$ bytes of data.

9.70.4.9 operator()()

```
lambda_reference operator() (
            const size_t i,
            const Monoid & monoid = Monoid() )  [inline]
```

Returns a lambda reference to an element of this sparse vector.

A lambda reference to an element of this vector is only valid when used inside a lambda function evaluated via grb::eWiseLambda. The lambda function is called for specific indices only– that is, the GraphBLAS implementation decides at which elements to dereference this container. Outside this scope the returned reference incurs undefined behaviour.

Warning

In particular, for the given index *i* by the lambda function, it shall be *illegal* to refer to indices relative to that *i*; including, but not limited to, $i + 1$, $i - 1$, et cetera.

Note

As a consequence, this function cannot be used to perform stencil or halo based operations.

If a previously non-existing entry of the vector is requested, a new nonzero is added at position *i* in this vector. The new element will have its initial value equal to the *identity* corresponding to the given monoid.

Warning

In parallel contexts the use of a returned lambda reference outside the context of an eWiseLambda will incur at least one of the following ill effects: it may

1. fail outright,
2. work on stale data,
3. work on incorrect data, or
4. incur high communication costs to guarantee correctness. In short, such usage causes undefined behaviour. Implementers are *not* advised to provide GAS-like functionality through this interface, as it invites bad programming practices and bad algorithm design decisions. This operator is instead intended to provide for generic BLAS1-type operations only.

Note

For I/O, use the iterator retrieved via cbegin() instead of relying on a lambda_reference.

Parameters

in	*i*	Which element to return a lambda reference of.
in	*monoid*	Under which generalised monoid to interpret the requested ith element of this vector.

Note

The *monoid* (or a ring) is required to be able to interpret a sparse vector. A user who is sure this vector is dense, or otherwise is able to ensure that the a lambda_reference will only be requested at elements where nonzeroes already exists, may refer to Vector::operator[ ],

Returns

A lambda reference to the element *i* of this vector.

Example.

See grb::eWiseLambda() for a practical and useful example.

Warning

There is no similar concept in the official GraphBLAS specs.

See also

lambda_reference For more details on the returned reference type.

grb::eWiseLambda For one legal way in which to use the returned lambda_reference.

9.70.4.10 operator=()

```
Vector< D, implementation, C > & operator= (
            Vector< D, implementation, C > && x )  [inline], [noexcept]
```

Move-from-temporary assignment.

Parameters

in,out	*x*	The temporary instance from which this instance shall take over its resources.

After a call to this function, *x* shall correspond to an empy vector.

Performance semantics

1. This move assignment completes in $\Theta(1)$ time.
2. This move assignment may not make system calls.
3. this move assignment moves $\Theta(1)$ data only.

9.70.4.11 operator[]()

```
lambda_reference operator[] (
            const size_t i )  [inline]
```

Returns a lambda reference to an element of this vector.

Warning

This functionality may only be used within the body of a lambda function that is passed into grb::eWiseLambda.

The user must ensure that the requested reference only corresponds to a pre-existing nonzero in this vector.

Warning

Requesting a nonzero entry at a coordinate where no nonzero exists results in undefined behaviour.

A lambda reference to an element of this vector is only valid when used inside a lambda function evaluated via grb::eWiseLambda. The lambda function is called for specific indices only– that is, ALP/GraphBLAS decides at which elements to dereference this container.

If such a lambda function dereferences multiple vectors, then the sparsity structure of the first vector passed as an argument to grb::eWiseLambda after the lambda function defines at which indices the vectors will be referenced. The user must ensure that all vectors dereferenced indeed have nonzeroes at every location this "leading vector" has a nonzero.

Warning

In particular, for the given index *i* by the lambda function, it shall be *illegal* to refer to indices relative to that *i*; including, but not limited to, $i+1$, $i-1$, et cetera.

Note

As a consequence, this function cannot be used to perform stencil or halo type operations.

For I/O purposes, use the iterator retrieved via cbegin() instead of relying on a lambda_reference.

Parameters

in	*i*	Which element to return a lambda reference of.

Returns

A lambda reference to the element *i* of this vector.

Example.

See grb::eWiseLambda for a practical and useful example.

See also

lambda_reference For more details on the returned reference type.

grb::eWiseLambda For one way to use the returned lambda_reference.

9.70.4.12 size()

```
RC size (
            T & size ) const  [inline]
```

Return the dimension of this vector.

Template Parameters

T	The integral output type.

Parameters

out	*size*	Where to store the size of this vector. The initial value is ignored.

Returns

grb::SUCCESS When the function call completes successfully.

Note

This function cannot fail.

Performance semantics

This function

1. contains $\Theta(1)$ work,
2. will not allocate new dynamic memory,
3. will take at most $\Theta(1)$ memory beyond the memory already used by the application before the call to this function.
4. will move at most $sizeof(T) + sizeof(size\_t)$ bytes of data.

The documentation for this class was generated from the following file:

- vector.hpp

9.71 zero< D > Class Template Reference

Standard identity for numerical addition.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.71.1 Detailed Description

template<typename D>
class grb::identities::zero< D >

Standard identity for numerical addition.

9.71.2 Member Function Documentation

9.71.2.1 value()

```
static constexpr D value ( )  [inline], [static], [constexpr]
```

Template Parameters

D	The domain of the value to return.

Returns

The identity under standard addition (i.e., 'zero').

The documentation for this class was generated from the following file:

- identities.hpp

9.72 zip< IN1, IN2, implementation > Class Template Reference

The zip operator that operators on keys as a left-hand input and values as a right hand input, producing a key-value `std::pair`.

```
#include <ops.hpp>
```

Inherits Operator< internal::zip< IN1, IN2, config::default_backend > >.

9.72.1 Detailed Description

template<typename IN1, typename IN2, enum Backend implementation = config::default_backend>
class grb::operators::zip< IN1, IN2, implementation >

The zip operator that operators on keys as a left-hand input and values as a right hand input, producing a key-value `std::pair`.

Template Parameters

IN1	The key type.
IN2	The value type.

The output domain is fixed to `std::pair< IN1, IN2 >`.

The documentation for this class was generated from the following file:

- ops.hpp

Chapter 10

File Documentation

10.1 graphblas.hpp File Reference

The main header to include in order to use the ALP/GraphBLAS API.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.
- namespace grb::algorithms::pregel

  The namespace for ALP/Pregel algorithms.
- namespace grb::interfaces

  The namespace for programming APIs that automatically translate to ALP/GraphBLAS.

Macros

- #define _GRB_BSP1D_BACKEND

  Which ALP/GraphBLAS backend the BSP1D backend should use for computations within a single user process.
- #define _GRB_NO_EXCEPTIONS

  Define this macro to turn off reliance on standard C++ exceptions.
- #define _GRB_NO_LIBNUMA

  Define this macro to disable the dependence on libnuma.
- #define _GRB_NO_STDIO

  Define this macro to turn off standard input/output support.
- #define _GRB_WITH_LPF

  Define this macro to compile with LPF support.

10.1.1 Detailed Description

The main header to include in order to use the ALP/GraphBLAS API.

Author

A. N. Yzelman.

Date

8th of August, 2016.

10.1.2 Macro Definition Documentation

10.1.2.1 _GRB_BSP1D_BACKEND

```
#define _GRB_BSP1D_BACKEND
```

Which ALP/GraphBLAS backend the BSP1D backend should use for computations within a single user process.

The ALP/GraphBLAS compiler wrapper sets this value automatically depending on the choice of backend– compare, e.g., the grb::BSP1D backend versus the grb::hybrid backend.

10.1.2.2 _GRB_NO_EXCEPTIONS

```
#define _GRB_NO_EXCEPTIONS
```

Define this macro to turn off reliance on standard C++ exceptions.

Deprecated Support for this macro is being phased out.

Note

Its intended use is to support ALP/GraphBLAS deployments on platforms that do not support C++ exceptions, such as some older Android SDK applications.

Warning

The safe usage of ALP/GraphBLAS while exceptions are disabled relies, at present, on the inspection of internal states and the usage of internal functions. We have no standardised exception-free way of using ALP/GraphBLAS at present and have no plans to (continue and/or extend) support for it.

10.1.2.3 _GRB_NO_LIBNUMA

```
#define _GRB_NO_LIBNUMA
```

Define this macro to disable the dependence on libnuma.

Warning

Defining this macro is discouraged and not tested thoroughly.

Note

The CMake bootstrap treats libnuma as a non-optional dependence.

10.1.2.4 _GRB_NO_STDIO

```
#define _GRB_NO_STDIO
```

Define this macro to turn off standard input/output support.

Warning

This macro has only been fully supported within the grb::banshee backend, where neither standard `iostream` nor `stdio.h` were available. If support through the full ALP implementation would be useful, please raise an issue through GitHub or Gitee so that we may consider and plan for supporting this macro more fully.

10.1.2.5 _GRB_WITH_LPF

```
#define _GRB_WITH_LPF
```

Define this macro to compile with LPF support.

Note

The CMake bootstrap automatically defines this flag when a valid LPF installation is found. This flag is also defined by the ALP/GraphBLAS compiler wrapper whenever an LPF-enabled backend is selected.

10.2 graphblas.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10 *
11 * Unless required by applicable law or agreed to in writing, software
12 * distributed under the License is distributed on an "AS IS" BASIS,
13 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 * See the License for the specific language governing permissions and
15 * limitations under the License.
16 */
17
370 #ifdef __DOXYGEN__
371
379 #define _GRB_NO_LIBNUMA
380
388 #define _GRB_NO_PINNING
389
400 #define _GRB_NO_STDIO
401
417 #define _GRB_NO_EXCEPTIONS
418
426 #define _GRB_WITH_LPF
427
436 #define _GRB_BACKEND reference
437
444 #define _GRB_BSP1D_BACKEND
445
452 namespace grb {
453
457     namespace algorithms {
458
462         namespace pregel {}
463
464     }
465
470     namespace interfaces {}
471
472 }
473
474 #endif // end "#ifdef __DOXYGEN__"
475
476 #ifndef _H_GRAPHBLAS
477 #define _H_GRAPHBLAS
478
479 // load active configuration
480 #include <graphblas/config.hpp> //defines _GRB_BACKEND and _WITH_BSP
481
482 // collects the user-level includes
483 // the order of these includes matter--
484 //    do not modify without proper consideration!
485
486 // First include all algebraic structures, which have the benefit of not
487 // depending on anything else
488 #include <graphblas/ops.hpp>
489 #include <graphblas/monoid.hpp>
490 #include <graphblas/semiring.hpp>
491
492 // Then include containers. If containers rely on ALP/GraphBLAS primitives that
493 // are defined as free functions, then container implementations must forward-
494 // declare those.
495 #include <graphblas/vector.hpp>
496 #include <graphblas/matrix.hpp>
497
498 // The aforementioned forward declarations must be in sync with the
499 // declarations of the user primitives defined as free functions in the below.
500 // The below relies on both algebraic structures/relations as well as container
501 // definitions. By maintaining the current order, these do not require forward
502 // declarations.
503 #include <graphblas/io.hpp>
504 #include <graphblas/benchmark.hpp>
505 #include <graphblas/blas0.hpp>
506 #include <graphblas/blas1.hpp>
507 #include <graphblas/blas2.hpp>
508 #include <graphblas/blas3.hpp>
509 #include <graphblas/collectives.hpp>
510 #include <graphblas/exec.hpp>
```

```
511 #include <graphblas/init.hpp>
512 #include <graphblas/ops.hpp>
513 #include <graphblas/pinnedvector.hpp>
514 #include <graphblas/properties.hpp>
515 #include <graphblas/spmd.hpp>
516
517 #ifdef _GRB_WITH_LPF
518  // collects various BSP utilities
519  #include <graphblas/bsp/spmd.hpp>
520 #endif
521
522 #endif // end "_H_GRAPHBLAS"
523
```

10.3 bicgstab.hpp File Reference

Implements the BiCGstab algorithm.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template< Descriptor descr = descriptors::no_operation, typename IOType , typename NonzeroType , typename InputType , typename ResidualType , class Semiring = Semiring< operators::add< InputType, InputType, InputType >, operators::mul< IOType, NonzeroType, InputType >, identities::zero, identities::one >, class Minus = operators::subtract< ResidualType >, class Divide = operators::divide< ResidualType >>

  RC bicgstab (grb::Vector< IOType > &x, const grb::Matrix< NonzeroType > &A, const grb::Vector< InputType > &b, const size_t max_iterations, ResidualType tol, size_t &iterations, ResidualType &residual, Vector< InputType > &r, Vector< InputType > &rhat, Vector< InputType > &p, Vector< InputType > &v, Vector< InputType > &s, Vector< InputType > &t, const Semiring &semiring=Semiring(), const Minus &minus=Minus(), const Divide &divide=Divide())

  Solves a linear system $b = Ax$ with x unknown by using the bi-conjugate gradient (bi-CG) stabilised method; i.e., BiCGstab.

10.3.1 Detailed Description

Implements the BiCGstab algorithm.

Author

A. N. Yzelman

Date

15th of February, 2022

Implementation time

To be taken with a pinch of salt, as it is highly subjective:

- 50 minutes, excluding error handling, documentation, and testing.
- 10 minutes to get it to compile, once the smoke test was generated.
- 15 minutes to incorporate proper error handling plus printing of warnings and errors.

10.4 bicgstab.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
35 #ifndef _H_GRB_ALGORITHMS_BICGSTAB
36 #define _H_GRB_ALGORITHMS_BICGSTAB
37
38 #include <graphblas.hpp>
39
40 #include <iostream>
41 #include <type_traits>
42
43 #ifdef _DEBUG
44  #include <cmath> // for sqrt, making the silent assumption that ResidualType
45                   // is a supported type for it
46 #endif
47
48
49 namespace grb {
50
51     namespace algorithms {
52
154         template<
155             Descriptor descr = descriptors::no_operation,
156             typename IOType, typename NonzeroType, typename InputType,
157             typename ResidualType,
158             class Semiring = Semiring<
159                 operators::add< InputType, InputType, InputType >,
160                 operators::mul< IOType, NonzeroType, InputType >,
161                 identities::zero, identities::one
162             >,
163             class Minus = operators::subtract< ResidualType >,
164             class Divide = operators::divide< ResidualType >
165         >
166         RC bicgstab(
167             grb::Vector< IOType > &x,
168             const grb::Matrix< NonzeroType > &A,
169             const grb::Vector< InputType > &b,
170             const size_t max_iterations,
171             ResidualType tol,
172             size_t &iterations,
173             ResidualType &residual,
174             Vector< InputType > &r,
175             Vector< InputType > &rhat,
176             Vector< InputType > &p,
177             Vector< InputType > &v,
178             Vector< InputType > &s,
179             Vector< InputType > &t,
180             const Semiring &semiring = Semiring(),
181             const Minus &minus = Minus(),
182             const Divide &divide = Divide()
183         ) {
184             // static checks
185             static_assert( !( descr & descriptors::no_casting ) || (
186                     std::is_same< IOType, NonzeroType >::value &&
187                     std::is_same< IOType, InputType >::value &&
188                     std::is_same< IOType, ResidualType >::value
189                 ), "no_casting descriptor was set but containers with differing domains "
190                 "were given."
191             );
192             static_assert( !( descr & descriptors::no_casting ) || (
193                     std::is_same< NonzeroType, typename Semiring::D1 >::value &&
194                     std::is_same< IOType, typename Semiring::D2 >::value &&
195                     std::is_same< InputType, typename Semiring::D3 >::value &&
196                     std::is_same< InputType, typename Semiring::D4 >::value
197                 ), "no_casting descriptor was set, but semiring has incompatible domains "
198                 "with the given containers."
199             );
200             static_assert( !( descr & descriptors::no_casting ) || (
```

```
201                     std::is_same< InputType, typename Minus::D1 >::value &&
202                     std::is_same< InputType, typename Minus::D2 >::value &&
203                     std::is_same< InputType, typename Minus::D3 >::value
204                 ), "no_casting descriptor was set, but given minus operator has "
205                 "incompatible domains with the given containers."
206             );
207             static_assert( !( descr & descriptors::no_casting ) || (
208                     std::is_same< ResidualType, typename Divide::D1 >::value &&
209                     std::is_same< ResidualType, typename Divide::D2 >::value &&
210                     std::is_same< ResidualType, typename Divide::D3 >::value
211                 ), "no_casting descriptor was set, but given divide operator has "
212                 "incompatible domains with the given tolerance type."
213             );
214             static_assert( std::is_floating_point< ResidualType >::value,
215                 "Require floating-point residual type."
216             );
217
218 #ifdef _DEBUG
219             std::cout « "Entering bicgstab; "
220                 « "tol = " « tol « ", "
221                 « "max_iterations = " « max_iterations « "\n";
222 #endif
223
224             // descriptor for indiciating dense computations
225             constexpr Descriptor dense_descr = descr | descriptors::dense;
226
227             // get an alias to zero and one in case 1 and 0 can't cast properly
228             const ResidualType zero = semiring.template getZero< ResidualType >();
229             const ResidualType one  = semiring.template getOne< ResidualType >();
230
231             // dynamic checks, sizes:
232             const size_t n = nrows( A );
233             if( n != ncols( A ) ) {
234                 return MISMATCH;
235             }
236             if( n != size( x ) ) {
237                 return MISMATCH;
238             }
239             if( n != size( b ) ) {
240                 return MISMATCH;
241             }
242             if( n != size( r ) || n != size( rhat ) || n != size( p ) ||
243                 n != size( p ) || n != size( s ) || n != size( t )
244             ) {
245                 return MISMATCH;
246             }
247
248             // dynamic checks, capacity:
249             if( n != capacity( x ) ) {
250                 return ILLEGAL;
251             }
252             if( n != capacity( r ) || n != capacity( rhat ) || n != capacity( p ) ||
253                 n != capacity( p ) || n != capacity( s ) || n != capacity( t )
254             ) {
255                 return ILLEGAL;
256             }
257
258             // dynamic checks, others:
259             if( tol <= zero ) {
260                 return ILLEGAL;
261             }
262
263 #ifdef _DEBUG
264             std::cout « "\t dynamic run-time error checking passed\n";
265 #endif
266
267             // prelude
268             ResidualType b_norm_squared = zero;
269             RC ret = dot< dense_descr >( b_norm_squared, b, b, semiring );
270             if( ret ) {
271                 std::cerr « "Error: BiCGstab encountered \"" « toString(ret)
272                     « "\" during computation of the norm of b\n";
273                 return ret;
274             }
275
276             // make it so that we do not need to take square roots when detecting
277             // convergence
278             tol *= tol;
279             tol *= b_norm_squared;
280 #ifdef _DEBUG
281             std::cout « "Effective squared relative tolerance is " « tol « "\n";
282 #endif
283
284             // ensure that x is structurally dense
285             if( nnz( x ) != n ) {
286                 ret = grb::set< descriptors::invert_mask | descriptors::structural >(
287                     x, x, zero
```

```
288                 );
289                 assert( nnz( x ) == n );
290             }
291 
292             // compute residual (squared), taking into account that b may be sparse
293             residual = zero;
294             ret = ret ? ret : set( t, zero );                                   // t = Ax
295             ret = ret ? ret : mxv< dense_descr >( t, A, x, semiring );
296             assert( nnz( t ) == n );
297             ret = ret ? ret : set( r, zero );                               // r = b - Ax
298             ret = ret ? ret : foldl( r, b, semiring.getAdditiveMonoid() );
299             assert( nnz( r ) == n );
300             ret = ret ? ret : foldl< dense_descr >( r, t, minus );
301             ret = ret ? ret : dot< dense_descr >( residual, r, r, semiring ); // residual
302 
303             // check for prelude error
304             if( ret ) {
305                 std::cerr « "Error: BiCGstab encountered \"" « toString(ret)
306                     « "\" during prelude\n";
307                 return ret;
308             }
309 
310             // check if the guess was good enough
311             if( residual < tol ) {
312                 return SUCCESS;
313             }
314 
315 #ifdef _DEBUG
316             std::cout « "\t prelude completed\n";
317 #endif
318 
319             // start iterations
320             ret = ret ? ret : set( rhat, r );
321             ret = ret ? ret : set( p, zero );
322             ret = ret ? ret : set( v, zero );
323             ResidualType rho, rho_old, alpha, beta, omega, temp;
324             rho_old = alpha = omega = one;
325             iterations = 0;
326 
327             for( ; ret == SUCCESS && iterations < max_iterations; ++iterations ) {
328 
329 #ifdef _DEBUG
330                 std::cout « "\t iteration " « iterations « " starts\n";
331 #endif
332 
333                 // rho = ( rhat, r )
334                 rho = zero;
335                 ret = ret ? ret : dot< dense_descr >( rho, rhat, r, semiring );
336 #ifdef _DEBUG
337                 std::cout « "\t\t rho = " « rho « "\n";
338 #endif
339                 if( ret == SUCCESS && rho == zero ) {
340                     std::cerr « "Error: BiCGstab detects r at iteration " « iterations «
341                         " is orthogonal to r-hat\n";
342                     return FAILED;
343                 }
344 
345                 // beta = (rho / rho_old) * (alpha / omega)
346                 ret = ret ? ret : apply( beta, rho, rho_old, divide );
347                 ret = ret ? ret : apply( temp, alpha, omega, divide );
348                 ret = ret ? ret : foldl( beta, temp, semiring.getMultiplicativeOperator() );
349 #ifdef _DEBUG
350                 std::cout « "\t\t beta = " « beta « "\n";
351 #endif
352 
353                 // p = r + beta ( p - omega * v )
354                 ret = ret ? ret : eWiseLambda(
355                     [&r,beta,&p,&v,omega,&semiring,&minus] (const size_t i) {
356                         InputType tmp;
357                         apply( tmp, omega, v[i], semiring.getMultiplicativeOperator() );
358                         foldl( p[ i ], tmp, minus );
359                         foldr( beta, p[ i ], semiring.getMultiplicativeOperator() );
360                         foldr( r[ i ], p[ i ], semiring.getAdditiveOperator() );
361                     }, v, p, r
362                 );
363 
364                 // v = Ap
365                 ret = ret ? ret : set( v, zero );
366                 ret = ret ? ret : mxv< dense_descr >( v, A, p, semiring );
367 
368                 // alpha = rho / (rhat, v)
369                 alpha = zero;
370                 ret = ret ? ret : dot< dense_descr >( alpha, rhat, v, semiring );
371                 if( alpha == zero ) {
372                     std::cerr « "Error: BiCGstab detects rhat is orthogonal to v=Ap "
373                         « "at iteration " « iterations « ".\n";
374                     return FAILED;
```

```
375                 }
376                 ret = ret ? ret : foldr( rho, alpha, divide );
377 #ifdef _DEBUG
378                 std::cout « "\t\t alpha = " « alpha « "\n";
379 #endif
380
381                 // x += alpha * p is post-poned to either the pre-stabilisation exit, or
382                 // after the stabilisation step
383                 //ret = ret ? ret : eWiseMul( x, alpha, p, semiring );
384
385                 // s = r - alpha * v
386                 {
387                     ResidualType minus_alpha = zero;
388                     ret = ret ? ret : foldl( minus_alpha, alpha, minus );
389                     ret = ret ? ret : set( s, r );
390                     ret = ret ? ret : eWiseMul< dense_descr >( s, minus_alpha, v, semiring );
391                 }
392
393                 // check residual
394                 residual = zero;
395                 ret = ret ? ret : dot< dense_descr >( residual, s, s, semiring );
396                 assert( residual > zero );
397 #ifdef _DEBUG
398                 std::cout « "\t\t running residual, pre-stabilisation: " « sqrt(residual)
399                     « "\n";
400 #endif
401                 if( ret == SUCCESS && residual < tol ) {
402                     // update result (x += alpha * p) and exit
403                     ret = eWiseMul< dense_descr >( x, alpha, p, semiring );
404                     return ret;
405                 }
406
407                 // t = As
408                 ret = ret ? ret : set( t, zero );
409                 ret = ret ? ret : mxv< dense_descr >( t, A, s, semiring );
410
411                 // omega = (t, s) / (t, t);
412                 omega = temp = zero;
413                 ret = ret ? ret : dot< dense_descr >( temp, t, s, semiring );
414 #ifdef _DEBUG
415                 std::cout « "\t\t (t, s) = " « temp « "\n";
416 #endif
417                 if( ret == SUCCESS && temp == zero ) {
418                     std::cerr « "Error: BiCGstab detects As at iteration " « iterations «
419                         " is orthogonal to s\n";
420                     return FAILED;
421                 }
422                 ret = ret ? ret : dot< dense_descr >( omega, t, t, semiring );
423 #ifdef _DEBUG
424                 std::cout « "\t\t (t, t) = " « omega « "\n";
425 #endif
426                 assert( omega > zero );
427                 ret = ret ? ret : foldr( temp, omega, divide );
428 #ifdef _DEBUG
429                 std::cout « "\t\t omega = " « omega « "\n";
430 #endif
431
432                 // x += alpha * p + omega * s
433                 ret = ret ? ret : eWiseMul< dense_descr >( x, alpha, p, semiring );
434                 ret = ret ? ret : eWiseMul< dense_descr >( x, omega, s, semiring );
435
436                 // r = s - omega * t
437                 {
438                     ResidualType minus_omega = zero;
439                     ret = ret ? ret : foldl( minus_omega, omega, minus );
440                     ret = ret ? ret : set( r, s );
441                     ret = ret ? ret : eWiseMul< dense_descr >( r, minus_omega, t, semiring );
442                 }
443
444                 // check residual
445                 residual = zero;
446                 ret = ret ? ret : dot< dense_descr >( residual, r, r, semiring );
447                 assert( residual > zero );
448 #ifdef _DEBUG
449                 std::cout « "\t\t running residual, post-stabilisation: "
450                     « sqrt(residual) « ". "
451                     « "Residual squared: " « residual « ".\n";
452 #endif
453                 if( ret == SUCCESS ) {
454                        if( residual < tol ) { return SUCCESS; }
455
456                     // go to next iteration
457                     rho_old = rho;
458                 }
459             }
460
461             if( ret == SUCCESS ) {
```

```
462                 // if we are here, then we did not detect convergence
463                 std::cerr « "Warning: call to BiCGstab did not converge within "
464                     « max_iterations « " iterations. Squared two-norm of the running "
465                     « "residual is " « residual « ". "
466                     « "Target residual squared: " « tol « ".\n";
467                 return FAILED;
468             } else {
469                 // if we are here, we exited due to an ALP error code
470                 std::cerr « "Error: BiCGstab encountered error \"" « toString(ret)
471                     « "\" while iterating to " « iterations « ", ";
472                 if( iterations == max_iterations ) {
473                     std::cerr « "which also is the maximum number of iterations.\n";
474                 } else {
475                     std::cerr « "which is below the maximum number of iterations of "
476                         « max_iterations « "\n";
477                 }
478                 return ret;
479             }
480         }
481
482     }
483 }
484
485 #endif // end _H_GRB_ALGORITHMS_BICGSTAB
486
```

10.5 conjugate_gradient.hpp File Reference

Implements the CG algorithm.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr = descriptors::no_operation, typename IOType , typename ResidualType , typename NonzeroType , typename InputType , class Ring = Semiring< grb::operators::add< IOType >, grb::operators::mul< IOType >, grb::identities::zero, grb::identities::one >, class Minus = operators::subtract< IOType >, class Divide = operators::divide< IOType >>
  grb::RC conjugate_gradient (grb::Vector< IOType > &x, const grb::Matrix< NonzeroType > &A, const grb::Vector< InputType > &b, const size_t max_iterations, ResidualType tol, size_t &iterations, ResidualType &residual, grb::Vector< IOType > &r, grb::Vector< IOType > &u, grb::Vector< IOType > &temp, const Ring &ring=Ring(), const Minus &minus=Minus(), const Divide &divide=Divide())

  Solves a linear system $b = Ax$ with x unknown by the Conjugate Gradients (CG) method on general fields.

10.5.1 Detailed Description

Implements the CG algorithm.

Author

Aristeidis Mastoras

10.6 conjugate_gradient.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
26 #ifndef _H_GRB_ALGORITHMS_CONJUGATE_GRADIENT
27 #define _H_GRB_ALGORITHMS_CONJUGATE_GRADIENT
28
29 #include <cstdio>
30 #include <complex>
31
32 #include <graphblas.hpp>
33 #include <graphblas/utils/iscomplex.hpp>
34
35
36 namespace grb {
37
38     namespace algorithms {
39
147         template< Descriptor descr = descriptors::no_operation,
148             typename IOType,
149             typename ResidualType,
150             typename NonzeroType,
151             typename InputType,
152             class Ring = Semiring<
153                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
154                 grb::identities::zero, grb::identities::one
155             >,
156             class Minus = operators::subtract< IOType >,
157             class Divide = operators::divide< IOType >
158         >
159         grb::RC conjugate_gradient(
160             grb::Vector< IOType > &x,
161             const grb::Matrix< NonzeroType > &A,
162             const grb::Vector< InputType > &b,
163             const size_t max_iterations,
164             ResidualType tol,
165             size_t &iterations,
166             ResidualType &residual,
167             grb::Vector< IOType > &r,
168             grb::Vector< IOType > &u,
169             grb::Vector< IOType > &temp,
170             const Ring &ring = Ring(),
171             const Minus &minus = Minus(),
172             const Divide &divide = Divide()
173         ) {
174             // static checks
175             static_assert( std::is_floating_point< ResidualType >::value,
176                 "Can only use the CG algorithm with floating-point residual "
177                 "types." ); // unless some different norm were used: issue #89
178             static_assert( !( descr & descriptors::no_casting ) || (
179                     std::is_same< IOType, ResidualType >::value &&
180                     std::is_same< IOType, NonzeroType >::value &&
181                     std::is_same< IOType, InputType >::value
182                 ), "One or more of the provided containers have differing element types "
183                 "while the no-casting descriptor has been supplied"
184             );
185             static_assert( !( descr & descriptors::no_casting ) || (
186                     std::is_same< NonzeroType, typename Ring::D1 >::value &&
187                     std::is_same< IOType, typename Ring::D2 >::value &&
188                     std::is_same< InputType, typename Ring::D3 >::value &&
189                     std::is_same< InputType, typename Ring::D4 >::value
190                 ), "no_casting descriptor was set, but semiring has incompatible domains "
191                 "with the given containers."
192             );
193             static_assert( !( descr & descriptors::no_casting ) || (
194                     std::is_same< InputType, typename Minus::D1 >::value &&
195                     std::is_same< InputType, typename Minus::D2 >::value &&
196                     std::is_same< InputType, typename Minus::D3 >::value
197                 ), "no_casting descriptor was set, but given minus operator has "
```

```
198             "incompatible domains with the given containers."
199         );
200         static_assert( !( descr & descriptors::no_casting ) || (
201                 std::is_same< ResidualType, typename Divide::D1 >::value &&
202                 std::is_same< ResidualType, typename Divide::D2 >::value &&
203                 std::is_same< ResidualType, typename Divide::D3 >::value
204             ), "no_casting descriptor was set, but given divide operator has "
205             "incompatible domains with the given tolerance type."
206         );
207         static_assert( std::is_floating_point< ResidualType >::value,
208             "Require floating-point residual type."
209         );
210 
211         constexpr const Descriptor descr_dense = descr | descriptors::dense;
212         const ResidualType zero_residual = ring.template getZero< ResidualType >();
213         const IOType zero = ring.template getZero< IOType >();
214         const size_t n = grb::ncols( A );
215 
216         // dynamic checks
217         {
218             const size_t m = grb::nrows( A );
219             if( size( x ) != n ) {
220                 return MISMATCH;
221             }
222             if( size( b ) != m ) {
223                 return MISMATCH;
224             }
225             if( size( r ) != n || size( u ) != n || size( temp ) != n ) {
226                 std::cerr « "Error: provided workspace vectors are not of the correct "
227                     « "length.\n";
228                 return MISMATCH;
229             }
230             if( m != n ) {
231                 std::cerr « "Warning: grb::algorithms::conjugate_gradient requires "
232                     « "square input matrices, but a non-square input matrix was "
233                     « "given instead.\n";
234                 return ILLEGAL;
235             }
236 
237             // capacities
238             if( capacity( x ) != n ) {
239                 return ILLEGAL;
240             }
241             if( capacity( r ) != n || capacity( u ) != n || capacity( temp ) != n ) {
242                 return ILLEGAL;
243             }
244 
245             // others
246             if( tol <= zero_residual ) {
247                 std::cerr « "Error: tolerance input to CG must be strictly positive\n";
248                 return ILLEGAL;
249             }
250         }
251 
252         // set pure output fields to neutral defaults
253         iterations = 0;
254         residual = std::numeric_limits< double >::infinity();
255 
256         // trivial shortcuts
257         if( max_iterations == 0 ) {
258             return FAILED;
259         }
260 
261         // make x and b structurally dense (if not already) so that the remainder
262         // algorithm can safely use the dense descriptor for faster operations
263         {
264             RC rc = SUCCESS;
265             if( nnz( x ) != n ) {
266                 rc = set< descriptors::invert_mask | descriptors::structural >(
267                     x, x, zero
268                 );
269             }
270             if( rc != SUCCESS ) {
271                 return rc;
272             }
273             assert( nnz( x ) == n );
274         }
275 
276         IOType sigma, bnorm, alpha, beta;
277 
278         // temp = 0
279         grb::RC ret = grb::set( temp, 0 );
280         assert( ret == SUCCESS );
281 
282         // temp = A * x
283         ret = ret ? ret : grb::mxv< descr_dense >( temp, A, x, ring );
284         assert( ret == SUCCESS );
```

```
285
286             // r = b - temp;
287             ret = ret ? ret : grb::set( r, zero );
288             ret = ret ? ret : grb::foldl( r, b, ring.getAdditiveMonoid() );
289             assert( nnz( r ) == n );
290             assert( nnz( temp ) == n );
291             ret = ret ? ret : grb::foldl< descr_dense >( r, temp, minus );
292             assert( ret == SUCCESS );
293             assert( nnz( r ) == n );
294
295             // u = r;
296             ret = ret ? ret : grb::set( u, r );
297             assert( ret == SUCCESS );
298
299             // sigma = r' * r;
300             sigma = zero;
301             if( grb::utils::is_complex< IOType >::value ) {
302                 ret = ret ? ret : grb::eWiseLambda( [&temp,&r]( const size_t i ) {
303                         temp[ i ] = grb::utils::is_complex< IOType >::conjugate( r[ i ] );
304                     }, temp
305                 );
306                 ret = ret ? ret : grb::dot< descr_dense >( sigma, temp, r, ring );
307             } else {
308                 ret = ret ? ret : grb::dot< descr_dense >( sigma, r, r, ring );
309             }
310
311             assert( ret == SUCCESS );
312
313             // bnorm = b' * b;
314             bnorm = zero;
315             if( grb::utils::is_complex< IOType >::value ) {
316                 ret = ret ? ret : grb::eWiseLambda( [&temp,&b]( const size_t i ) {
317                         temp[ i ] = grb::utils::is_complex< IOType >::conjugate( b[ i ] );
318                     }, temp
319                 );
320                 ret = ret ? ret : grb::dot< descr_dense >( bnorm, temp, b, ring );
321             } else {
322                 ret = ret ? ret : grb::dot< descr_dense >( bnorm, b, b, ring );
323             }
324             assert( ret == SUCCESS );
325
326             if( ret == SUCCESS ) {
327                 tol *= sqrt( grb::utils::is_complex< IOType >::modulus( bnorm ) );
328             }
329
330             size_t iter = 0;
331
332             do {
333                 assert( iter < max_iterations );
334                 (void) ++iter;
335
336                 // temp = 0
337                 ret = ret ? ret : grb::set( temp, 0 );
338                 assert( ret == SUCCESS );
339
340                 // temp = A * u;
341                 ret = ret ? ret : grb::mxv< descr_dense >( temp, A, u, ring );
342                 assert( ret == SUCCESS );
343
344                 // beta = u' * temp
345                 beta = zero;
346                 if( grb::utils::is_complex< IOType >::value ) {
347                     ret = ret ? ret : grb::eWiseLambda( [&u]( const size_t i ) {
348                             u[ i ] = grb::utils::is_complex< IOType >::conjugate( u[ i ] );
349                         }, u
350                     );
351                 }
352                 ret = ret ? ret : grb::dot< descr_dense >( beta, temp, u, ring );
353                 if( grb::utils::is_complex< IOType >::value ) {
354                     ret = ret ? ret : grb::eWiseLambda( [&u]( const size_t i ) {
355                             u[ i ] = grb::utils::is_complex< IOType >::conjugate( u[ i ] );
356                         }, u
357                     );
358                 }
359                 assert( ret == SUCCESS );
360
361                 // alpha = sigma / beta;
362                 ret = ret ? ret : grb::apply( alpha, sigma, beta, divide );
363                 assert( ret == SUCCESS );
364
365                 // x = x + alpha * u;
366                 ret = ret ? ret : grb::eWiseMul< descr_dense >( x, alpha, u, ring );
367                 assert( ret == SUCCESS );
368
369                 // temp = alpha .* temp
370                 // Warning: operator-based foldr requires temp be dense
371                 ret = ret ? ret : grb::foldr( alpha, temp, ring.getMultiplicativeMonoid() );
```

```
372                 assert( ret == SUCCESS );
373
374                 // r = r - temp;
375                 ret = ret ? ret : grb::foldl< descr_dense >( r, temp, minus );
376                 assert( ret == SUCCESS );
377
378                 // beta = r' * r;
379                 beta = zero;
380                 if( grb::utils::is_complex< IOType >::value ) {
381                     ret = ret ? ret : grb::eWiseLambda( [&temp,&r]( const size_t i ) {
382                             temp[ i ] = grb::utils::is_complex< IOType >::conjugate( r[ i ] );
383                         }, temp
384                     );
385                     ret = ret ? ret : grb::dot< descr_dense >( beta, temp, r, ring );
386                 } else {
387                     ret = ret ? ret : grb::dot< descr_dense >( beta, r, r, ring );
388                 }
389                 residual = grb::utils::is_complex< IOType >::modulus( beta );
390                 assert( ret == SUCCESS );
391
392                 if( ret == SUCCESS ) {
393                     if( sqrt( residual ) < tol || iter >= max_iterations ) {
394                         break;
395                     }
396                 }
397
398                 // alpha = beta / sigma;
399                 ret = ret ? ret : grb::apply( alpha, beta, sigma, divide );
400                 assert( ret == SUCCESS );
401
402                 // temp = r + alpha * u;
403                 ret = ret ? ret : grb::set( temp, r );
404                 assert( ret == SUCCESS );
405                 ret = ret ? ret : grb::eWiseMul< descr_dense >( temp, alpha, u, ring );
406                 assert( ret == SUCCESS );
407                 assert( nnz( temp ) == size( temp ) );
408
409                 // u = temp
410                 std::swap( u, temp );
411
412                 sigma = beta;
413             } while( ret == SUCCESS );
414
415             // output that is independent of error code
416             iterations = iter;
417
418             // return correct error code
419             if( ret == SUCCESS ) {
420                 if( sqrt( residual ) >= tol ) {
421                     // did not converge within iterations
422                     return FAILED;
423                 }
424             }
425             return ret;
426         }
427
428     } // namespace algorithms
429
430 } // end namespace grb
431
432 #endif // end _H_GRB_ALGORITHMS_CONJUGATE_GRADIENT
433
```

10.7 cosine_similarity.hpp File Reference

Implements cosine simularity.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType1 , typename InputType2 , class Ring , class Division = grb::operators::divide< typename Ring::D3, typename Ring::D3, typename Ring::D4 >>
  RC cosine_similarity (OutputType &similarity, const Vector< InputType1 > &x, const Vector< InputType2 > &y, const Ring &ring=Ring(), const Division &div=Division())

  Computes the cosine similarity.

10.7.1 Detailed Description

Implements cosine simularity.

Author

: A. N. Yzelman.

Date

: 13th of December, 2017.

10.8 cosine_similarity.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_COSSIM
28 #define _H_GRB_COSSIM
29
30 #include <graphblas.hpp>
31 #include <graphblas/algorithms/norm.hpp>
32
33
34 #define NO_CAST_ASSERT( x, y, z )                                                                    \
35     static_assert( x,                                                                                \
36         "\n\n"                                                                                       \
37         "*****************************************************************************************\n" \
38         "*     ERROR      | " y " " z ".\n"                                                         \
39         "*****************************************************************************************\n" \
40         "* Possible fix 1 | Remove no_casting from the template parameters in this call to " y ".   \n" \
41         "* Possible fix 2 | For all mismatches in the domains of input and output parameters w.r.t. \n" \
42         "*                  the semiring domains, as specified in the documentation of the function.\n" \
43         "*                  supply an input argument of the expected type instead.\n"                \
44         "* Possible fix 3 | Provide a compatible semiring where all domains match those of the input\n" \
45         "*                  parameters, as specified in the documentation of the function.\n"        \
46         "*****************************************************************************************\n"
     );
47
48
49 namespace grb {
50
51     namespace algorithms {
52
106         template<
```

```
107             Descriptor descr = descriptors::no_operation,
108             typename OutputType,
109             typename InputType1,
110             typename InputType2,
111             class Ring,
112             class Division = grb::operators::divide<
113                 typename Ring::D3, typename Ring::D3, typename Ring::D4
114             >
115         >
116         RC cosine_similarity(
117             OutputType &similarity,
118             const Vector< InputType1 > &x, const Vector< InputType2 > &y,
119             const Ring &ring = Ring(), const Division &div = Division()
120         ) {
121             static_assert( std::is_floating_point< OutputType >::value,
122                 "Cosine similarity requires a floating-point output type." );
123
124             // static sanity checks
125             NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) ||
126                     std::is_same< InputType1, typename Ring::D1 >::value
127                 ), "grb::algorithms::cosine_similarity",
128                 "called with a left-hand vector value type that does not match the "
129                 "first domain of the given semiring" );
130             NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) ||
131                     std::is_same< InputType2, typename Ring::D2 >::value
132                 ), "grb::algorithms::cosine_similarity",
133                 "called with a right-hand vector value type that does not match "
134                 "the second domain of the given semiring" );
135             NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) ||
136                     std::is_same< OutputType, typename Ring::D4 >::value
137                 ), "grb::algorithms::cosine_similarity",
138                 "called with an output vector value type that does not match the "
139                 "fourth domain of the given semiring" );
140             NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) ||
141                     std::is_same< typename Ring::D3, typename Ring::D4 >::value
142                 ), "grb::algorithms::cosine_similarity",
143                 "called with a semiring that has unequal additive input domains" );
144
145             const size_t n = size( x );
146
147             // run-time sanity checks
148             if( n != size( y ) ) {
149                 return MISMATCH;
150             }
151
152             // check whether inputs are dense
153             const bool dense = nnz( x ) == n && nnz( y ) == n;
154
155             // set return code
156             RC rc = SUCCESS;
157
158             // compute-- choose method depending on we can stream once or need to stream
159             // multiple times
160             OutputType nominator, denominator;
161             nominator = denominator = ring.template getZero< OutputType >();
162             if( dense && grb::Properties<>::writableCaptured ) {
163                 // lambda works, so we can stream each vector precisely once:
164                 OutputType norm1, norm2;
165                 norm1 = norm2 = ring.template getZero< OutputType >();
166                 rc = grb::eWiseLambda(
167                     [ &x, &y, &nominator, &norm1, &norm2, &ring ]( const size_t i ) {
168                         const auto &mul = ring.getMultiplicativeOperator();
169                         const auto &add = ring.getAdditiveOperator();
170                         OutputType temp;
171                         (void) grb::apply( temp, x[ i ], y[ i ], mul );
172                         (void) grb::foldl( nominator, temp, add );
173                         (void) grb::apply( temp, x[ i ], x[ i ], mul );
174                         (void) grb::foldl( norm1, temp, add );
175                         (void) grb::apply( temp, y[ i ], y[ i ], mul );
176                         (void) grb::foldl( norm2, temp, add );
177                     }, x, y
178                 );
179                 denominator = sqrt( norm1 ) * sqrt( norm2 );
180             } else {
181                 // cannot stream each vector once, stream each one twice instead using
182                 // standard grb functions
183                 rc = grb::norm2( nominator, x, ring );
184                 if( rc == SUCCESS ) {
185                     rc = grb::norm2( denominator, y, ring );
186                 }
187                 if( rc == SUCCESS ) {
188                     rc = grb::foldl( denominator, nominator,
189                         ring.getMultiplicativeOperator() );
190                 }
191                 if( rc == SUCCESS ) {
192                     rc = grb::dot( nominator, x, y, ring );
193                 }
```

```
194             }
195
196             // accumulate
197             if( rc == SUCCESS ) {
198                 // catch zeroes
199                 if( denominator == ring.template getZero() ) {
200                     return ILLEGAL;
201                 }
202                 if( nominator == ring.template getZero() ) {
203                     return ILLEGAL;
204                 }
205                 rc = grb::apply( similarity, nominator, denominator, div );
206             }
207
208             // done
209             return rc;
210         }
211
212     } // end namespace algorithms
213
214 } // end namespace grb
215
216 #undef NO_CAST_ASSERT
217
218 #endif // end _H_GRB_COSSIM
219
```

10.9 kcore_decomposition.hpp File Reference

Implements the algebraic k-core decomposition algorithm by Li et al.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr = descriptors::no_operation, bool criticalSection = false, typename IOType , typename NZType >
  RC kcore_decomposition (const Matrix< NZType > &A, Vector< IOType > &core, Vector< IOType > &distances, Vector< IOType > &temp, Vector< IOType > &update, Vector< bool > &status, IOType &k)

  The k-core decomposition algorithm.

10.9.1 Detailed Description

Implements the algebraic k-core decomposition algorithm by Li et al.

Author

Anders Hansson

Date

January, 2023

10.10 kcore_decomposition.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
28 #ifndef _H_GRB_KCORE_DECOMPOSITION
29 #define _H_GRB_KCORE_DECOMPOSITION
30
31 #include <graphblas.hpp>
32
33
34 namespace grb {
35
36     namespace algorithms {
37
136         template<
137             Descriptor descr = descriptors::no_operation,
138             bool criticalSection = false,
139             typename IOType, typename NZType
140         >
141         RC kcore_decomposition(
142             const Matrix< NZType > &A,
143             Vector< IOType > &core,
144             Vector< IOType > &distances,
145             Vector< IOType > &temp,
146             Vector< IOType > &update,
147             Vector< bool >   &status,
148             IOType &k
149         ) {
150             // Add constants/expressions
151             Semiring<
152                 operators::add< IOType >, operators::mul< IOType >,
153                 identities::zero, identities::one
154             > ring;
155             Monoid<
156                 operators::logical_or< bool >,
157                 identities::logical_false
158             > lorMonoid;
159
160             // Runtime sanity checks
161             const size_t n = nrows(A);
162             {
163                 // Verify that A is square
164                 if( n != ncols( A )){
165                     return ILLEGAL;
166                 }
167                 // Verify sizes of vectors
168                 if( size( core ) != n ||
169                     size( distances ) != n ||
170                     size( temp ) != n ||
171                     size( update ) != n ||
172                     size( status ) != n
173                 ) {
174                     return MISMATCH;
175                 }
176                 // Verify capacity
177                 if( capacity( core ) != n ||
178                     capacity( distances ) != n ||
179                     capacity( temp ) != n ||
180                     capacity( update ) != n ||
181                     capacity( status ) != n
182                 ) {
183                     return ILLEGAL;
184                 }
185             }
186
187             // Initialise
188             IOType current_k = 0; // current coreness level
189
190             // Set initial values
```

```
191             RC ret = grb::SUCCESS;
192             ret = ret ? ret : set( temp, static_cast< IOType >( 1 ) );
193             ret = ret ? ret : set( distances,  static_cast< IOType >( 0 ) );
194             ret = ret ? ret : set( core,  static_cast< IOType >( 0 ) );
195             ret = ret ? ret : set( status, true );
196             ret = ret ? ret : clear( update );
197             assert( ret == SUCCESS );
198 
199             ret = ret ? ret : grb::mxv< descr | descriptors::dense >(
200                 distances, A, temp, ring );
201             assert( ret == SUCCESS );
202 
203             if( SUCCESS != ret ) {
204                 std::cerr « " Initialization of k-core decomposition failed with error "
205                     « grb::toString( ret ) « "\n";
206                 return ret;
207             }
208 
209             size_t count = 0;
210             while( count < n && SUCCESS == ret ) {
211                 bool flag = true;
212 
213                 // Update filter to exclude completed nodes
214                 ret = ret ? ret : set( update, status, status );
215 
216                 while( flag ) {
217                     flag = false;
218 
219                     // Update nodes in parallel
220                     if( criticalSection ) {
221                         ret = ret ? ret : clear( temp );
222                         ret = ret ? ret : eWiseLambda( [ &, current_k ]( const size_t i ) {
223                                 if( status[ i ] && distances[ i ] <= current_k ) {
224                                     core[ i ] = current_k;
225                                     // Remove node from checking
226                                     status[ i ] = false;
227                                     // Set update
228                                     flag = true;
229                                     #pragma omp critical
230                                     {
231                                         // Add node index to update neighbours
232                                         setElement( temp, 1, i );
233                                     }
234                                 }
235                             }, update,
236                             status, distances, core, temp
237                         );
238                         // WARN: even with the below, this variant does not auto-parallelise in
239                         //       the distributed-memory sense. The reason is a performance
240                         //       contract violation by the above critical section -- setElement
241                         //       should be a collective call, but its use from within eWiseLambda
242                         //       does not ensure a collective call. The result is that PANIC will
243                         //       at some point be returned.
244                         //ret = ret ? ret : collectives<>::allreduce( flag,
245                         //  lorMonoid.getOperator() );
246                     } else {
247                         ret = ret ? ret : eWiseApply( temp, status, distances, current_k,
248                             operators::leq< IOType >() );
249                         ret = ret ? ret : foldl( core, temp, current_k,
250                             operators::right_assign< IOType >() );
251                         ret = ret ? ret : foldl( status, temp, false,
252                             operators::right_assign< bool >() );
253                         ret = ret ? ret : foldl( flag, temp, lorMonoid );
254                         ret = ret ? ret : set( update, temp, 1 );
255                         if( ret == SUCCESS ) {
256                             std::swap( update, temp );
257                         }
258                     }
259                     assert( ret == SUCCESS );
260 
261                     if( ret == SUCCESS && flag ) {
262                         ret = clear( update );
263                         assert( ret == SUCCESS );
264 
265                         // Increase number of nodes completed
266                         count += nnz( temp );
267 
268                         // Get the neighbours of the updated nodes
269                         ret = ret ? ret : grb::mxv< descr >( update, A, temp, ring );
270                         assert( ret == SUCCESS );
271 
272                         // Decrease distances of the neighbours
273                         ret = ret ? ret : grb::eWiseApply( distances, distances, update,
274                             operators::subtract< IOType >() );
275                         assert( ret == SUCCESS );
276                     }
277                 }
```

```
278                 (void) ++current_k;
279             }
280
281             if( SUCCESS != ret ){
282                 std::cerr « " Excecution of k-core decomposition failed with error "
283                     « grb::toString(ret) « "\n";
284             } else {
285                 k = current_k;
286             }
287
288             return ret;
289         }
290
291     } // namespace algorithms
292
293 } // namespace grb
294
295 #endif // end _H_GRB_KCORE_DECOMPOSITION
296
```

10.11 kmeans.hpp File Reference

Implements k-means.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr = descriptors::no_operation, typename IOType = double, class Operator = operators::square_diff< IOType, IOType, IOType >>
  RC kmeans_iteration (Matrix< IOType > &K, Vector< std::pair< size_t, IOType > > &clusters_and_↩ distances, const Matrix< IOType > &X, const size_t max_iter=1000, const Operator &dist_op=Operator())

  The kmeans iteration given an initialisation.
- template<Descriptor descr = descriptors::no_operation, typename IOType = double, class Operator = operators::square_diff< IOType, IOType, IOType >>
  RC kpp_initialisation (Matrix< IOType > &K, const Matrix< IOType > &X, const Operator &dist_↩ op=Operator())

  a simple implementation of the k++ initialisation algorithm for kmeans

10.11.1 Detailed Description

Implements k-means.

The state of the algorithms defined within are *experimental*.

Author

Verner Vlacic

10.12 kmeans.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_KMEANS
28 #define _H_GRB_KMEANS
29
30 #include <chrono>
31 #include <random>
32
33 #include <assert.h>
34
35 #include <graphblas.hpp>
36
37
38 namespace grb {
39
40     namespace algorithms {
41
55         template<
56             Descriptor descr = descriptors::no_operation,
57             typename IOType = double,
58             class Operator = operators::square_diff< IOType, IOType, IOType >
59         >
60         RC kpp_initialisation(
61             Matrix< IOType > &K,
62             const Matrix< IOType > &X,
63             const Operator &dist_op = Operator()
64         ) {
65             // declare monoids and semirings
66             Monoid< grb::operators::add< IOType >, grb::identities::zero > add_monoid;
67             Monoid<
68                 grb::operators::min< IOType >,
69                 grb::identities::infinity
70             > min_monoid;
71             Semiring<
72                 grb::operators::add< IOType >,
73                 grb::operators::right_assign_if< bool, IOType, IOType >,
74                 grb::identities::zero, grb::identities::logical_true
75             > pattern_sum;
76
77             // runtime sanity checks: the row dimension of X should match the column
78             // dimension of K
79             if( ncols( K ) != nrows( X ) ) {
80                 return MISMATCH;
81             }
82
83             // running error code
84             RC ret = SUCCESS;
85
86             // get problem dimensions
87             const size_t n = ncols( X );
88             const size_t m = nrows( X );
89             const size_t k = nrows( K );
90
91             // declare vector of indices of columns of X selected as the initial
92             // centroids
93             Vector< size_t > selected_indices( k );
94
95             // declare column selection vector
96             Vector< bool > col_select( n );
97
98             // declare selected point
99             Vector< IOType > selected( m );
100
101              // declare vector of distances from the selected point
102              Vector< IOType > selected_distances( n );
103
104              // declare vector of minimum distances to all points selected so far
```

```
105             Vector< IOType > min_distances( n );
106             ret = ret ? ret : grb::set( min_distances,
107                 grb::identities::infinity< IOType >::value()
108             );
109
110             // generate first centroid by selecting a column of X uniformly at random
111
112             size_t i;
113             {
114                 const size_t seed_uniform =
115                     std::chrono::system_clock::now().time_since_epoch().count();
116                 std::default_random_engine random_generator( seed_uniform );
117                 std::uniform_int_distribution< size_t > uniform( 0, n - 1 );
118                 i = uniform( random_generator );
119             }
120
121             for( size_t l = 0; ret == SUCCESS && l < k; ++l ) {
122
123                 ret = grb::clear( col_select );
124                 ret = ret ? ret : grb::clear( selected );
125                 ret = ret ? ret : grb::clear( selected_distances );
126
127                 ret = ret ? ret : grb::setElement( selected_indices, i, l );
128
129                 ret = ret ? ret : grb::setElement( col_select, true, i );
130
131                 ret = ret ? ret : grb::vxm< grb::descriptors::transpose_matrix >(
132                     selected, col_select, X, pattern_sum );
133
134                 ret = ret ? ret : grb::vxm( selected_distances, selected, X, add_monoid,
135                     dist_op );
136
137                 ret = ret ? ret : grb::foldl( min_distances, selected_distances,
138                     min_monoid );
139
140                 // TODO the remaining part of the loop should be replaced with the alias
141                 //      algorithm
142
143                 IOType range = add_monoid.template getIdentity< IOType >();
144                 ret = ret ? ret : grb::foldl( range, min_distances, add_monoid );
145
146                 double sample = -1;
147                 if( ret == SUCCESS ) {
148                     {
149                         const size_t seed =
150                             std::chrono::system_clock::now().time_since_epoch().count();
151                         std::default_random_engine generator( seed );
152                         std::uniform_real_distribution< double > uniform( 0, 1 );
153                         sample = uniform( generator );
154                     }
155                     ret = grb::collectives<>::broadcast( sample, 0 );
156                 }
157                 assert( sample >= 0 );
158
159                 // The following is not standard ALP/GraphBLAS and does not work for P>1
160                 //    (TODO internal issue #320)
161                 if( ret == SUCCESS ) {
162                     assert( grb::spmd<>::nprocs() == 1 );
163                     IOType * const raw = internal::getRaw( selected_distances );
164                     IOType running_sum = 0;
165                     i = 0;
166                     do {
167                         running_sum += static_cast< double >( raw[ i ] ) / range;
168                     } while( running_sum < sample && ++i < n );
169                     i = ( i == n ) ? n - 1 : i;
170                 }
171             }
172
173             // create the matrix K by selecting the columns of X indexed by
174             // selected_indices
175
176             // declare pattern matrix
177             Matrix< void > M( k, n );
178             ret = ret ? ret : grb::resize( M, n );
179
180             if( ret == SUCCESS ) {
181                 auto converter = grb::utils::makeVectorToMatrixConverter< void, size_t >(
182                     selected_indices, [](const size_t &ind, const size_t &val ) {
183                         return std::make_pair( ind, val );
184                     }
185                 );
186                 ret = grb::buildMatrixUnique( M, converter.begin(), converter.end(),
187                     PARALLEL );
188             }
189
190             ret = ret ? ret : grb::mxm< descriptors::transpose_right >( K, M, X,
191                 pattern_sum, RESIZE );
```

```
192             ret = ret ? ret : grb::mxm< descriptors::transpose_right >( K, M, X,
193                 pattern_sum );
194
195             if( ret != SUCCESS ) {
196                 std::cout « "\tkpp finished with unexpected return code!" « std::endl;
197             }
198
199             return ret;
200         }
201
219         template<
220             Descriptor descr = descriptors::no_operation,
221             typename IOType = double,
222             class Operator = operators::square_diff< IOType, IOType, IOType >
223         >
224         RC kmeans_iteration(
225             Matrix< IOType > &K,
226             Vector< std::pair< size_t, IOType > > &clusters_and_distances,
227             const Matrix< IOType > &X,
228             const size_t max_iter = 1000,
229             const Operator &dist_op = Operator()
230         ) {
231             // declare monoids and semirings
232
233             typedef std::pair< size_t, IOType > indexIOType;
234
235             Monoid< grb::operators::add< IOType >, grb::identities::zero > add_monoid;
236
237             Monoid<
238                 grb::operators::argmin< size_t, IOType >,
239                 grb::identities::infinity
240             > argmin_monoid;
241
242             Monoid<
243                 grb::operators::logical_and< bool >,
244                 grb::identities::logical_true
245             > comparison_monoid;
246
247             Semiring<
248                 grb::operators::add< IOType >,
249                 grb::operators::right_assign_if< bool, IOType, IOType >,
250                 grb::identities::zero, grb::identities::logical_true
251             > pattern_sum;
252
253             Semiring<
254                 grb::operators::add< size_t >,
255                 grb::operators::right_assign_if< size_t, size_t, size_t >,
256                 grb::identities::zero, grb::identities::logical_true
257             > pattern_count;
258
259             // runtime sanity checks: the row dimension of X should match the column
260             // dimension of K
261             if( ncols( K ) != nrows( X ) ) {
262                 return MISMATCH;
263             }
264             if( size( clusters_and_distances ) != ncols( X ) ) {
265                 return MISMATCH;
266             }
267
268             // running error code
269             RC ret = SUCCESS;
270
271             // get problem dimensions
272             const size_t n = ncols( X );
273             const size_t m = nrows( X );
274             const size_t k = nrows( K );
275
276             // declare distance matrix
277             Matrix< IOType > Dist( k, n );
278
279             // declare and initialise labels vector and ones vectors
280             Vector< size_t > labels( k );
281             Vector< bool > n_ones( n ), m_ones( m );
282
283             ret = ret ? ret : grb::set< grb::descriptors::use_index >( labels, 0 );
284             ret = ret ? ret : grb::set( n_ones, true );
285             ret = ret ? ret : grb::set( m_ones, true );
286
287             // declare pattern matrix
288             Matrix< void > M( k, n );
289             ret = ret ? ret : grb::resize( M, n );
290
291             // declare the sizes vector
292             Vector< size_t > sizes( k );
293
294             // declare auxiliary vectors and matrices
295             Matrix< IOType > K_aux( k, m );
```

```
296             Matrix< size_t > V_aux( k, m );
297
298             // control variables
299             size_t iter = 0;
300             Vector< indexIOType > clusters_and_distances_prev( n );
301             bool converged;
302
303             do {
304                 (void) ++iter;
305
306                 ret = ret ? ret : grb::set( clusters_and_distances_prev,
307                     clusters_and_distances );
308
309                 ret = ret ? ret : mxm( Dist, K, X, add_monoid, dist_op, RESIZE );
310                 ret = ret ? ret : mxm( Dist, K, X, add_monoid, dist_op );
311
312                 ret = ret ? ret : vxm( clusters_and_distances, labels, Dist, argmin_monoid,
313                     operators::zip< size_t, IOType >() );
314
315                 auto converter = grb::utils::makeVectorToMatrixConverter<
316                     void, indexIOType
317                 > (
318                     clusters_and_distances,
319                     []( const size_t &ind, const indexIOType &pair ) {
320                         return std::make_pair( pair.first, ind );
321                     }
322                 );
323
324                 ret = ret ? ret : grb::buildMatrixUnique( M, converter.begin(),
325                     converter.end(), PARALLEL );
326
327                 ret = ret ? ret : grb::mxm< descriptors::transpose_right >( K_aux, M, X,
328                     pattern_sum, RESIZE );
329                 ret = ret ? ret : grb::mxm< descriptors::transpose_right >( K_aux, M, X,
330                     pattern_sum );
331
332                 ret = ret ? ret : grb::mxv( sizes, M, n_ones, pattern_count );
333
334                 ret = ret ? ret : grb::outer( V_aux, sizes, m_ones,
335                     operators::left_assign_if< IOType, bool, IOType >(), RESIZE );
336                 ret = ret ? ret : grb::outer( V_aux, sizes, m_ones,
337                     operators::left_assign_if< IOType, bool, IOType >() );
338
339                 ret = ret ? ret : eWiseApply( K, V_aux, K_aux,
340                     operators::divide_reverse< size_t, IOType, IOType >(), RESIZE );
341                 ret = ret ? ret : eWiseApply( K, V_aux, K_aux,
342                     operators::divide_reverse< size_t, IOType, IOType >() );
343
344                 converged = true;
345                 ret = ret ? ret : grb::dot(
346                     converged,
347                     clusters_and_distances_prev, clusters_and_distances,
348                     comparison_monoid,
349                     grb::operators::equal_first< indexIOType, indexIOType, bool >()
350                 );
351
352             } while( ret == SUCCESS && !converged && iter < max_iter );
353
354             if( iter == max_iter ) {
355                 std::cout « "\tkmeans reached maximum number of iterations!" « std::endl;
356                 return FAILED;
357             }
358             if( converged ) {
359                 std::cout « "\tkmeans converged successfully after " « iter
360                     « " iterations." « std::endl;
361                 return SUCCESS;
362             }
363
364             std::cout « "\tkmeans finished with unexpected return code!" « std::endl;
365             return ret;
366         }
367
368     } // namespace algorithms
369
370 } // namespace grb
371
372 #endif // end _H_GRB_KMEANS
373
```

10.13 knn.hpp File Reference

Implements the k-hop nearest neighbours from a given source vertex.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr, typename OutputType , typename InputType >
  RC knn (Vector< OutputType > &u, const Matrix< InputType > &A, const size_t source, const size_t k, Vector< bool > &buf1)

  Given a graph and a source vertex, indicates which vertices are contained within k hops.

10.13.1 Detailed Description

Implements the k-hop nearest neighbours from a given source vertex.

Author

A. N. Yzelman

Date

: 27th of April, 2017

10.14 knn.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_KNN
28 #define _H_GRB_KNN
29
30 #include "graphblas/algorithms/mpv.hpp"
31
32 #include <graphblas.hpp>
33
34
35 namespace grb {
36
37     namespace algorithms {
38
81         template< Descriptor descr, typename OutputType, typename InputType >
82         RC knn(
83             Vector< OutputType > &u, const Matrix< InputType > &A,
84             const size_t source, const size_t k,
```

```
85              Vector< bool > &buf1
86          ) {
87              // the nearest-neighbourhood ring
88              Semiring<
89                  operators::logical_or< bool >, operators::logical_and< bool >,
90                  identities::logical_false, identities::logical_true
91              > ring;
92
93              // check input
94              const size_t n = nrows( A );
95              if( n != ncols( A ) ) {
96                  return MISMATCH;
97              }
98              if( size( buf1 ) != n ) {
99                  return MISMATCH;
100             }
101             if( size( u ) != n ) {
102                 return MISMATCH;
103             }
104             if( capacity( u ) != n ) {
105                 return ILLEGAL;
106             }
107             if( capacity( buf1 ) != n ) {
108                 return ILLEGAL;
109             }
110
111             // prepare
112             RC ret = SUCCESS;
113             if( nnz( u ) != 0 ) {
114                 ret = clear( u );
115             }
116             if( nnz( buf1 ) != 0 ) {
117                 ret = ret ? ret : clear( buf1 );
118             }
119 #ifdef _DEBUG
120             std::cout « "grb::algorithms::knn called with source " « source « " "
121                 « "and k " « k « ".\n";
122 #endif
123             ret = ret ? ret : setElement( buf1, true, source );
124
125             // do sparse matrix powers on the given ring
126             if( ret == SUCCESS ) {
127                 if( descr & descriptors::transpose_matrix ) {
128                     ret = mpv< (descr | descriptors::add_identity) &
129                         ~( descriptors::transpose_matrix )
130                     >( u, A, k, buf1, buf1, ring );
131                 } else {
132                     ret = mpv< descr | descriptors::add_identity |
133                         descriptors::transpose_matrix
134                     >( u, A, k, buf1, buf1, ring );
135                 }
136             }
137
138             // done
139             return ret;
140         }
141
142     } // namespace algorithms
143
144 } // namespace grb
145
146 #endif
147
```

10.15 label.hpp File Reference

Implements label propagation.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<typename IOType >
  RC label (Vector< IOType > &out, const Vector< IOType > &y, const Matrix< IOType > &W, const size_t n, const size_t l, const size_t maxIterations=1000)

  The label propagation algorithm.

10.15.1 Detailed Description

Implements label propagation.

Author

J. M. Nash

Date

21st of March, 2017

10.16 label.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_LABEL
28 #define _H_GRB_LABEL
29
30 #include <iostream>
31
32 #include <graphblas.hpp>
33
34
35 namespace grb {
36
37     namespace algorithms {
38
39 #ifdef _DEBUG
40         constexpr size_t MaxPrinting = 20;
41         constexpr size_t MaxAnyPrinting = 100;
42
43         // take a vector and display with a message
44         static void printVector(
45             const Vector< double > &v, const std::string message
46         ) {
47             size_t zeros = 0;
48             size_t ones = 0;
49             size_t size = grb::size( v );
50             if( size > MaxAnyPrinting ) {
51                 return;
52             }
53             std::cerr « "\t " « message « ": ";
54             for( Vector< double >::const_iterator it = v.begin(); it != v.end(); ++it ) {
55                 const std::pair< size_t, double > iter = *it;
```

```
56                  const double val = iter.second;
57                  if( val < INFINITY ) {
58                      if( size > MaxPrinting ) {
59                          zeros += ( val == 0 ) ? 1 : 0;
60                          ones += ( val == 1 ) ? 1 : 0;
61                      } else {
62                          std::cerr « val « " ";
63                      }
64                  }
65              }
66              if( size > MaxPrinting ) {
67                  std::cerr « zeros « " zeros; " « ones « " ones.";
68              }
69              std::cerr « "\n";
70          }
71  #endif
72
122         template< typename IOType >
123         RC label(
124             Vector< IOType > &out,
125             const Vector< IOType > &y, const Matrix< IOType > &W,
126             const size_t n, const size_t l,
127             const size_t maxIterations = 1000
128         ) {
129             // label propagation vectors and matrices operate over the real domain
130             Semiring<
131                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
132                 grb::identities::zero, grb::identities::one
133             > reals;
134             grb::operators::not_equal< IOType, IOType, bool > notEqualOp;
135             grb::Monoid<
136                 grb::operators::logical_or< bool >,
137                 grb::identities::logical_false
138             > orMonoid;
139             const IOType zero = reals.template getZero< IOType >();
140
141             // dynamic checks
142             if( nrows( W ) != n || ncols( W ) != n ||
143                 size( y ) != n || size( out ) != n
144             ) {
145                 return ILLEGAL;
146             }
147             if( capacity( out ) != n ) {
148                 return ILLEGAL;
149             }
150             if( n == 0 ) {
151                 return SUCCESS;
152             }
153             if( l == 0 ) {
154                 return ILLEGAL;
155             }
156
157             const size_t s = spmd<>::pid();
158
159             // compute the diagonal matrix D from the weight matrix W
160             // we represent D as a vector so we can use it to generate the probabilities
161             // matrix P
162             Vector< IOType > multiplier( n );
163             RC ret = set( multiplier, static_cast< IOType >(1) ); // a vector of 1's
164
165             Vector< IOType > diagonals( n );
166             ret = ret ? ret : set( diagonals, zero );
167             ret = ret ? ret : mxv< descriptors::dense >(
168                 diagonals, W, multiplier, reals
169             ); // W*multiplier will sum each row
170 #ifdef _DEBUG
171             printVector( diagonals, "diagonals matrix in vector form" );
172 #endif
173
174             // compute the probabilistic transition matrix P as inverse of D * W
175             // use diagonals vector to directly compute probabilistic transition matrix P
176             // only the existing non-zero elements in W will map to P
177             // the inverse of D is represented via the inverse element in the diagonals
178             // vector
179             //
180             // update: the application of Dinv is now done within a lambda following the
181             //         mxv on the original matrix P
182
183             // make diagonals equal its inverse
184             ret = ret ? ret : eWiseLambda( [ &diagonals ]( const size_t i ) {
185                     diagonals[ i ] = 1.0 / diagonals[ i ];
186                 }, diagonals
187             );
188
189             // set up current and new solution functions
190             Vector< IOType > f( n );
191             Vector< IOType > fNext( n );
```

```
192             Vector< bool > mask( n );
193             for( size_t i = 0; ret == SUCCESS && i < l; ++i ) {
194                 ret = setElement( mask, true, i );
195             }
196 
197             // fix f = y for the input set of labels
198             ret = ret ? ret : set( f, y );
199 
200             // whether two successive solutions are different
201             // initially set to true so that the computation may begin
202             bool different = true;
203             // compute f as P*f
204             // main loop completes when function f is stable
205             size_t iter = 1;
206             while( ret == SUCCESS && different && iter < maxIterations ) {
207 
208 #ifdef _DEBUG
209                 if( n < MaxAnyPrinting ) {
210                     std::cerr « "\t iteration " « iter « "\n";
211                 }
212 
213                 // fNext = P*f
214                 std::cerr « "\t pre- set/mxv nnz( f ) = " « nnz( f ) « ", "
215                     « "fNext = " « nnz( fNext ) « "\n";
216 #endif
217                 ret = ret ? ret : set( fNext, zero );
218                 ret = ret ? ret : mxv( fNext, W, f, reals );
219 #ifdef _DEBUG
220                 std::cerr « "\t post-set/mxv nnz( f ) = " « nnz( f ) « ", "
221                     « "nnz( fNext ) = " « nnz( fNext ) « "\n";
222                 printVector( f, "Previous iteration solution" );
223                 printVector( fNext, "New iteration solution" );
224 #endif
225 
226                 // maintain the solution function in domain {0,1}
227                 // can use the masked variant of vector assign when available ?
228                 ret = ret ? ret : eWiseLambda( [ &fNext, &diagonals ]( const size_t i ) {
229                         fNext[ i ] = ( fNext[ i ] * diagonals[ i ] < 0.5 ? 0 : 1 );
230                     }, diagonals, fNext
231                 );
232 #ifdef _DEBUG
233                 printVector( fNext, "New iteration solution after threshold cutoff" );
234                 std::cerr « "\t pre-set  nnz( fNext ) = " « nnz( fNext ) « ", "
235                     « "nnz( mask ) = " « nnz( mask ) « "\n";
236 #endif
237                 // clamps the first l labelled nodes
238                 ret = ret ? ret : foldl(
239                     fNext, mask,
240                     f,
241                     grb::operators::right_assign< IOType >()
242                 );
243                 assert( ret == SUCCESS );
244 #ifdef _DEBUG
245                 std::cerr « "\t post-set nnz( fNext ) = " « nnz( fNext ) « "\n";
246                 printVector(
247                     fNext,
248                     "New iteration solution after threshold cutoff and clamping"
249                 );
250 #endif
251                 // test for stability
252                 different = false;
253                 ret = ret ? ret : dot( different, f, fNext, orMonoid, notEqualOp );
254 
255                 // update f for the next iteration
256 #ifdef _DEBUG
257                 std::cerr « "\t pre-set  nnz(f) = " « nnz( f ) « "\n";
258 #endif
259                 std::swap( f, fNext );
260 #ifdef _DEBUG
261                 std::cerr « "\t post-set nnz(f) = " « nnz( f ) « "\n";
262 #endif
263                 // go to next iteration
264                 (void) ++iter;
265             }
266 
267             if( ret == SUCCESS ) {
268                 if( different ) {
269                     if( s == 0 ) {
270                         std::cerr « "Info: label propagation did not converge after "
271                             « (iter-1) « " iterations\n";
272                     }
273                     return FAILED;
274                 } else {
275                     if( s == 0 ) {
276                         std::cerr « "Info: label propagation converged in "
277                             « (iter-1) « " iterations\n";
278                     }
```

```
279                     std::swap( out, f );
280                     return SUCCESS;
281                 }
282             }
283
284             // done
285             if( s == 0 ) {
286                 std::cerr « "Warning: label propagation exiting with " « toString(ret)
287                     « "\n";
288             }
289             return ret;
290         }
291
292     } // namespace algorithms
293
294 } // namespace grb
295
296 #endif // end _H_GRB_LABEL
297
```

10.17 mpv.hpp File Reference

Implements the matrix powers kernel $y = A^k x$ over arbitrary semirings.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr, class Ring , typename IOType , typename InputType >
  RC mpv (Vector< IOType > &u, const Matrix< InputType > &A, const size_t k, const Vector< IOType > &v, Vector< IOType > &temp, const Ring &ring)

  The matrix powers kernel.

10.17.1 Detailed Description

Implements the matrix powers kernel $y = A^k x$ over arbitrary semirings.

Author

A. N. Yzelman

Date

30th of March 2017

10.18 mpv.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_ALGORITHMS_MPV
28 #define _H_GRB_ALGORITHMS_MPV
29
30 #include <graphblas.hpp>
31
32
33 namespace grb {
34
35     namespace algorithms {
36
93         template< Descriptor descr, class Ring, typename IOType, typename InputType >
94         RC mpv(
95             Vector< IOType > &u,
96             const Matrix< InputType > &A, const size_t k,
97             const Vector< IOType > &v,
98             Vector< IOType > &temp,
99             const Ring &ring
100          ) {
101             static_assert( !(descr & descriptors::no_casting) ||
102                     (std::is_same< IOType, typename Ring::D4 >::value &&
103                         std::is_same< InputType, typename Ring::D2 >::value &&
104                         std::is_same< IOType, typename Ring::D1 >::value &&
105                         std::is_same< IOType, typename Ring::D3 >::value
106                     ),
107                 "grb::mpv : some containers were passed with element types that do not"
108                 "match the given semiring domains."
109             );
110
111             // runtime check
112             const size_t n = nrows( A );
113             if( n != ncols( A ) ) {
114                 return ILLEGAL;
115             }
116             if( size( u ) != n || n != size( v ) ) {
117                 return MISMATCH;
118             }
119             if( size( temp ) != n ) {
120                 return MISMATCH;
121             }
122             if( capacity( u ) != n ) {
123                 return ILLEGAL;
124             }
125             if( capacity( temp ) != n ) {
126                 return ILLEGAL;
127             }
128             // catch trivial case
129             if( k == 0 ) {
130                 return set< descr >( u, v );
131             }
132             // otherwise, do at least one multiplication
133 #ifdef _DEBUG
134             std::cout « "init: input vector nonzeroes is " « grb::nnz( v ) « ".\n";
135 #endif
136             RC ret = mxv< descr >( u, A, v, ring );
137             if( k == 1 ) {
138                 return ret;
139             }
140             // do any remaining multiplications using a temporary output vector
141             bool copy;
142             ret = ret ? ret : clear( temp );
143             for( size_t iterate = 1; ret == SUCCESS && iterate < k; iterate += 2 ) {
144                 // multiply with output into temporary
145                 copy = true;
146 #ifdef _DEBUG
147                 std::cout « "up: input vector nonzeroes is " « grb::nnz( u ) « "\n";
```

```
148 #endif
149                 ret = mxv< descr >( temp, A, u, ring );
150                 // check if this was the final multiplication
151                 assert( iterate <= k );
152                 if( iterate + 1 == k || ret != SUCCESS ) {
153                     break;
154                 }
155                 // multiply with output into u
156                 copy = false;
157 #ifdef _DEBUG
158                 std::cout « "down: input vector nonzeroes is " « grb::nnz( temp ) « "\n";
159 #endif
160                 ret = mxv< descr >( u, A, temp, ring );
161             }
162 
163             // swap u and temp, if required
164             if( ret == SUCCESS && copy ) {
165                 std::swap( u, temp );
166             }
167 
168             // done
169             return ret;
170         }
171 
172     } // namespace algorithms
173 
174 } // namespace grb
175 
176 #endif // end "_H_GRB_ALGORITHMS_MPV"
177 
```

10.19 norm.hpp File Reference

Implements the 2-norm.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr = descriptors::no_operation, class Ring , typename InputType , typename OutputType , Backend backend, typename Coords >
  RC norm2 (OutputType &x, const Vector< InputType, backend, Coords > &y, const Ring &ring=Ring(), const typename std::enable_if< std::is_floating_point< OutputType >::value, void >::type *const =nullptr)

  Provides a generic implementation of the 2-norm computation.

10.19.1 Detailed Description

Implements the 2-norm.

Author

A. N. Yzelman

Date

17th of March 2022

10.20 norm.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
34 #ifndef _H_GRB_ALGORITHMS_NORM
35 #define _H_GRB_ALGORITHMS_NORM
36
37 #include <graphblas.hpp>
38
39 #include <cmath> // for std::sqrt
40
41
42 namespace grb {
43
44     namespace algorithms {
45
71         template<
72             Descriptor descr = descriptors::no_operation, class Ring,
73             typename InputType, typename OutputType,
74             Backend backend, typename Coords
75         >
76         RC norm2( OutputType &x,
77             const Vector< InputType, backend, Coords > &y,
78             const Ring &ring = Ring(),
79             const typename std::enable_if<
80                 std::is_floating_point< OutputType >::value,
81             void >::type * const = nullptr
82         ) {
83             RC ret = grb::dot< descr >( x, y, y, ring );
84             if( ret == SUCCESS ) {
85                 x = sqrt( x );
86             }
87             return ret;
88         }
89
90     }
91 }
92
93 #endif // end "_H_GRB_ALGORITHMS_NORM"
94
```

10.21 pregel_connected_components.hpp File Reference

Implements the (strongly) connected components algorithm over undirected graphs using the ALP/Pregel interface.

Classes

- struct ConnectedComponents< VertexIDType >

  A vertex-centric Connected Components algorithm.
- struct ConnectedComponents< VertexIDType >::Data

  This vertex-centric Connected Components algorithm does not require any algorithm parameters.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.
- namespace grb::algorithms::pregel

  The namespace for ALP/Pregel algorithms.

10.21.1 Detailed Description

Implements the (strongly) connected components algorithm over undirected graphs using the ALP/Pregel interface.

Author

: A. N. Yzelman.

10.22 pregel_connected_components.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_PREGEL_CONNECTEDCOMPONENTS
28 #define _H_GRB_PREGEL_CONNECTEDCOMPONENTS
29
30 #include <graphblas/interfaces/pregel.hpp>
31
32
33 namespace grb {
34
35     namespace algorithms {
36
37         namespace pregel {
38
47             template< typename VertexIDType >
48             struct ConnectedComponents {
49
54                 struct Data {};
55
81                 static void program(
82                     VertexIDType &current_max_ID,
83                     const VertexIDType &incoming_message,
84                     VertexIDType &outgoing_message,
85                     const Data &parameters,
86                     grb::interfaces::PregelState &pregel
87                 ) {
88                     (void) parameters;
89                     if( pregel.round > 0 ) {
90                         if( pregel.indegree == 0 ) {
91                             pregel.voteToHalt = true;
92                         } else if( current_max_ID < incoming_message ) {
93                             current_max_ID = incoming_message;
94                         } else {
95                             pregel.voteToHalt = true;
96                         }
```

```
97                      }
98                      if( pregel.outdegree > 0 ) {
99                          outgoing_message = current_max_ID;
100                     } else {
101                         pregel.voteToHalt = true;
102                     }
103                 }
104
124                 template< typename PregelType >
125                 static grb::RC execute(
126                     grb::interfaces::Pregel< PregelType > &pregel,
127                     grb::Vector< VertexIDType > &group_ids,
128                     const size_t max_steps = 0,
129                     size_t * const steps_taken = nullptr
130                 ) {
131                     const size_t n = pregel.num_vertices();
132                     if( grb::size( group_ids ) != n ) {
133                         return MISMATCH;
134                     }
135
136                     grb::RC ret = grb::set< grb::descriptors::use_index >( group_ids, 1 );
137                     if( ret != SUCCESS ) {
138                         return ret;
139                     }
140
141                     grb::Vector< VertexIDType > in( n );
142                     grb::Vector< VertexIDType > out( n );
143                     grb::Vector< VertexIDType > out_buffer = interfaces::config::out_sparsify
144                         ? grb::Vector< VertexIDType >( n )
145                         : grb::Vector< VertexIDType >( 0 );
146
147                     size_t steps;
148
149                     ret = pregel.template execute<
150                         grb::operators::max< VertexIDType >,
151                         grb::identities::negative_infinity
152                     > (
153                         program,
154                         group_ids,
155                         Data(),
156                         in, out,
157                         steps,
158                         out_buffer,
159                         max_steps
160                     );
161
162                     if( ret == grb::SUCCESS && steps_taken != nullptr ) {
163                         *steps_taken = steps;
164                     }
165
166                     return ret;
167                 }
168
169             };
170
171         } //end namespace 'grb::algorithms::pregel'
172
173     } // end namespace "grb::algorithms"
174
175 } // end namespace "grb"
176
177 #endif
178
```

10.23 pregel_pagerank.hpp File Reference

Implements a traditional vertex-centric page ranking algorithm using ALP/Pregel.

Classes

- struct PageRank< IOType, localConverge >::Data

  The algorithm parameters.
- struct PageRank< IOType, localConverge >

  A Pregel-style PageRank-like algorithm.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.
- namespace grb::algorithms::pregel

  The namespace for ALP/Pregel algorithms.

10.23.1 Detailed Description

Implements a traditional vertex-centric page ranking algorithm using ALP/Pregel.

Author

A. N. Yzelman

10.24 pregel_pagerank.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_PREGEL_PAGERANK
28 #define _H_GRB_PREGEL_PAGERANK
29
30 #include <graphblas/interfaces/pregel.hpp>
31
32
33 namespace grb {
34
35     namespace algorithms {
36
37         namespace pregel {
38
53             template< typename IOType, bool localConverge >
54             struct PageRank {
55
59                 struct Data {
60
64                     IOType alpha = 0.15;
65
69                     IOType tolerance = 0.00001;
70
71                 };
72
87                 static void program(
88                     IOType &current_score,
89                     const IOType &incoming_message,
90                     IOType &outgoing_message,
91                     const Data &parameters,
92                     grb::interfaces::PregelState &pregel
93                 ) {
94                     // initialise
95                     if( pregel.round == 0 ) {
96                         current_score = static_cast< IOType >( 1 );
```

```
97                      }
98
99 #ifdef _DEBUG
100                     // when in debug mode, probably one does not wish to track the state of
101                     // each vertex individually, hence we include a simple guard by default:
102                     const bool dbg = pregel.vertexID == 0;
103                     if( dbg ) {
104                         std::cout « "ID: " « pregel.vertexID « "\n"
105                             « "\t active: " « pregel.active « "\n"
106                             « "\t round: " « pregel.round « "\n"
107                             « "\t previous score: " « current_score « "\n"
108                             « "\t incoming message: " « incoming_message « "\n";
109                     }
110 #endif
111
112                     // compute
113                     if( pregel.round > 0 ) {
114                         const IOType old_score = current_score;
115                         current_score = parameters.alpha +
116                             (static_cast< IOType >(1) - parameters.alpha) * incoming_message;
117                         if( fabs(current_score-old_score) < parameters.tolerance ) {
118 #ifdef _DEBUG
119                             std::cout « "\t\t vertex " « pregel.vertexID « " converged\n";
120 #endif
121                             if( localConverge ) {
122                                 pregel.active = false;
123                             } else {
124                                 pregel.voteToHalt = true;
125                             }
126                         }
127                     }
128
129                     // broadcast
130                     if( pregel.outdegree > 0 ) {
131                         outgoing_message =
132                             current_score /
133                             static_cast< IOType >(pregel.outdegree);
134                     }
135
136 #ifdef _DEBUG
137                     if( dbg ) {
138                         std::cout « "\t current score: " « current_score « "\n"
139                             « "\t voteToHalt: " « pregel.voteToHalt « "\n"
140                             « "\t outgoing message: " « outgoing_message « "\n";
141                     }
142 #endif
143
144                 }
145
182                 template< typename PregelType >
183                 static grb::RC execute(
184                     grb::interfaces::Pregel< PregelType > &pregel,
185                     grb::Vector< IOType > &scores,
186                     size_t &steps_taken,
187                     const Data &parameters = Data(),
188                     const size_t max_steps = 0
189                 ) {
190                     const size_t n = pregel.num_vertices();
191                     if( grb::size( scores ) != n ) {
192                         return MISMATCH;
193                     }
194
195                     grb::Vector< IOType > in( n );
196                     grb::Vector< IOType > out( n );
197                     grb::Vector< IOType > out_buffer = interfaces::config::out_sparsify
198                         ? grb::Vector< IOType >( n )
199                         : grb::Vector< IOType >( 0 );
200
201                     return pregel.template execute<
202                             grb::operators::add< IOType >,
203                             grb::identities::zero
204                         > (
205                             program,
206                             scores,
207                             parameters,
208                             in, out,
209                             steps_taken,
210                             out_buffer,
211                             max_steps
212                         );
213                 }
214
215             };
216
217         } //end namespace 'grb::algorithms::pregel'
218
219     } // end namespace "grb::algorithms"
```

```
220
221 } // end namespace "grb"
222
223 #endif
224
```

10.25 simple_pagerank.hpp File Reference

Implements the canonical PageRank algorithm by Brin and Page.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr = descriptors::no_operation, typename IOType , typename NonzeroT >
  RC simple_pagerank (Vector< IOType > &pr, const Matrix< NonzeroT > &L, Vector< IOType > &pr_next, Vector< IOType > &pr_nextnext, Vector< IOType > &row_sum, const IOType alpha=0.85, const IOType conv=0.0000001, const size_t max=1000, size_t *const iterations=nullptr, double *const quality=nullptr)

  The canonical PageRank algorithm.

10.25.1 Detailed Description

Implements the canonical PageRank algorithm by Brin and Page.

Author

A. N. Yzelman

Date

: 21st of March, 2017

10.26 simple_pagerank.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_PAGERANK
28 #define _H_GRB_PAGERANK
29
30 #include <graphblas.hpp>
31
32 #ifndef _GRB_NO_STDIO
33 #include <iostream>
34 #endif
35
36
37 namespace grb {
38
39     namespace algorithms {
40
130         template<
131             Descriptor descr = descriptors::no_operation,
132             typename IOType, typename NonzeroT
133         >
134         RC simple_pagerank(
135             Vector< IOType > &pr,
136             const Matrix< NonzeroT > &L,
137             Vector< IOType > &pr_next,
138             Vector< IOType > &pr_nextnext,
139             Vector< IOType > &row_sum,
140             const IOType alpha = 0.85,
141             const IOType conv = 0.0000001,
142             const size_t max = 1000,
143             size_t * const iterations = nullptr,
144             double * const quality = nullptr
145         ) {
146             grb::Monoid<
147                 grb::operators::add< IOType >,
148                 grb::identities::zero
149             > addM;
150             grb::Semiring<
151                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
152                 grb::identities::zero, grb::identities::one
153             > realRing;
154 #ifdef _DEBUG
155             const auto s = spmd<>::pid();
156 #endif
157             const size_t n = nrows( L );
158             const IOType zero = realRing.template getZero< IOType >();
159
160             // runtime sanity checks
161             {
162                 if( n != ncols( L ) ) {
163                     return ILLEGAL;
164                 }
165                 if( size( pr ) != n ) {
166                     return MISMATCH;
167                 }
168                 if( size( pr ) != n ) {
169                     return MISMATCH;
170                 }
171                 if( size( pr_next ) != n ||
172                     size( pr_nextnext ) != n ||
173                     size( row_sum ) != n
174                 ) {
175                     return MISMATCH;
176                 }
177                 if( capacity( pr ) != n ) {
178                     return ILLEGAL;
179                 }
180                 if( capacity( pr_next ) != n ||
```

```
181                     capacity( pr_nextnext ) != n ||
182                     capacity( row_sum ) != n
183                 ) {
184                     return ILLEGAL;
185                 }
186                 // alpha must be within 0 and 1 (both exclusive)
187                 if( alpha <= 0 || alpha >= 1 ) {
188                     return ILLEGAL;
189                 }
190                 // max must be larger than 0
191                 if( max <= 0 ) {
192                     return ILLEGAL;
193                 }
194             }
195 
196             // running error code
197             RC ret = SUCCESS;
198 
199             // make initial guess if the user did not make one
200             if( nnz( pr ) != n ) {
201                 ret = set( pr, static_cast< IOType >( 1 ) / static_cast< IOType >( n ) );
202                 assert( ret == SUCCESS );
203             }
204 
205             // initialise all temporary vectors to default dense values
206             ret = ret ? ret : set( pr_nextnext, zero );
207             assert( ret == SUCCESS );
208 
209             // calculate row sums
210             Semiring<
211                 operators::add< IOType >,
212                 operators::left_assign_if< IOType, bool, IOType >,
213                 identities::zero,
214                 identities::logical_true
215             > pattern_ring;
216 
217             ret = ret ? ret : set( pr_next, 1 ); // abuses pr_next as temporary vector
218             ret = ret ? ret : set( row_sum, 0 );
219             ret = ret ? ret :
220                 vxm< descr | descriptors::dense | descriptors::transpose_matrix >(
221                     row_sum, pr_next, L, pattern_ring
222                 );
223             // pr_next is now free for further use
224             assert( ret == SUCCESS );
225 
226 #ifdef _DEBUG
227             std::cout « "Prelude to iteration 0:\n";
228             (void) eWiseLambda(
229                 [ &row_sum, &pr_next, &pr ]( const size_t i ) {
230                     #pragma omp critical
231                     {
232                         std::cout « i « ": " « row_sum[ i ] « "\t" « pr[ i ] « "\t"
233                             « pr_next[ i ] « "\n";
234                     }
235                 },
236                 pr, pr_next, row_sum );
237 #endif
238 
239             // calculate in-place the inverse of row sums, and keep zero if dangling
240             // this eWiseLambda is always supported since alpha is read-only
241             ret = ret ? ret : eWiseLambda(
242                     [ &row_sum, &alpha, &zero ]( const size_t i ) {
243                         assert( row_sum[ i ] >= zero );
244                         if( row_sum[ i ] > zero ) {
245                             row_sum[ i ] = alpha / row_sum[ i ];
246                         }
247                     },
248                     row_sum
249                 );
250                 assert( ret == SUCCESS );
251 
252 #ifdef _DEBUG
253             std::cout « "Row sum array:\n";
254             (void) eWiseLambda(
255                 [ &row_sum ]( const size_t i ) {
256                     #pragma omp critical
257                     std::cout « i « ": " « row_sum[ i ] « "\n";
258                 }, row_sum
259             );
260 #endif
261 
262             // some control variables
263             size_t iter = 0; //#iterations, initalise to zero
264             IOType dangling = zero; // used for caching the contribution of random jumps
265                                     // from dangling nodes
266             IOType residual = zero; // declare residual (computed inside the do-while
267                                     // loop)
```

```
268
269             // main loop
270             do {
271                 // reset iteration-local values
272                 residual = dangling = 0;
273
274 #ifdef _DEBUG
275                 std::cout « "Current PR array:\n";
276                 (void) eWiseLambda(
277                         [ &pr ]( const size_t i ) {
278                             #pragma omp critical
279                             if( i < 8 ) {
280                                 std::cout « i « ": " « pr[ i ] « "\n";
281                             }
282                         }, pr
283                     );
284 #endif
285
286                 // calculate dangling factor and do scaling
287                 if( ret == SUCCESS ) {
288                     // can we reduce via lambdas?
289                     if( Properties<>::writableCaptured ) {
290                         // yes we can, so save one unnecessary stream on pr
291                         ret = eWiseLambda(
292                                 [ &pr_next, &row_sum, &dangling, &pr ]( const size_t i ) {
293                                     // calculate dangling contribution
294                                     if( row_sum[ i ] == 0 ) {
295                                         dangling += pr[ i ];
296                                         pr_next[ i ] = 0;
297                                     } else {
298                                         // pre-scale input
299                                         pr_next[ i ] = pr[ i ] * row_sum[ i ];
300                                     }
301                                 }, row_sum, pr, pr_next
302                             );
303                         // allreduce dangling factor
304                         assert( ret == SUCCESS );
305                         ret = ret ? ret : grb::collectives<>::allreduce(
306                             dangling,
307                             grb::operators::add< double >()
308                         );
309                         assert( ret == SUCCESS );
310                     } else {
311                         // otherwise we have to handle the reduction separately
312                         ret = foldl< grb::descriptors::invert_mask >(
313                             dangling, pr, row_sum, addM
314                         );
315                         assert( ret == SUCCESS );
316
317                         // separately from the element-wise multiplication here
318                         ret = ret ? ret : set( pr_next, 0 );
319                         ret = ret ? ret : eWiseApply(
320                                 pr_next, pr, row_sum,
321                                 grb::operators::mul< double >()
322                             );
323                         assert( ret == SUCCESS );
324                     }
325                 }
326
327 #if defined _DEBUG && defined _GRB_WITH_LPF
328                 for( size_t dbg = 0; dbg < spmd<>::nprocs(); ++dbg ) {
329                     const auto s = spmd<>::pid();
330                     if( dbg == s ) {
331                         std::cout « "Next PR array (under construction):\n";
332                         eWiseLambda(
333                             [ &pr_next, s ]( const size_t i ) {
334                                 #pragma omp critical
335                                 if( i < 10 ) {
336                                     std::cout « i « ", " « s « ": " « pr_next[ i ] « "\n";
337                                 }
338                             },
339                             pr_next );
340                     }
341                     spmd<>::sync();
342                 }
343 #endif
344
345                 if( ret == SUCCESS ) {
346 #ifdef _DEBUG
347                     std::cout « s « ": dangling (1) = " « dangling « "\n";
348 #endif
349
350                     // complete dangling factor
351                     dangling = ( alpha * dangling + 1 - alpha ) / static_cast< IOType >( n );
352
353 #ifdef _DEBUG
354                     std::cout « s « ": dangling (2) = " « dangling « "\n";
```

```
355 #endif
356                 }
357
358                 // multiply with row-normalised link matrix (no change to dangling rows)
359                 // note that the later eWiseLambda requires the output be dense
360                 ret = ret ? ret : set( pr_nextnext, 0 ); assert( ret == SUCCESS );
361                 ret = ret ? ret : vxm< descr >( pr_nextnext, pr_next, L, realRing );
362                 assert( ret == SUCCESS );
363                 assert( n == grb::nnz( pr_nextnext ) );
364
365 #if defined _DEBUG && defined _GRB_WITH_LPF
366                 for( size_t dbg = 0; dbg < spmd<>::nprocs(); ++dbg ) {
367                     if( dbg == s ) {
368                         std::cout « s « ": nextnext PR array (after vxm):\n";
369                         (void) eWiseLambda(
370                             [ &pr_nextnext, s ]( const size_t i ) {
371                                 #pragma omp critical
372                                 if( i < 10 )
373                                     std::cout « i « ", " « s « ": " « pr_nextnext[ i ] « "\n";
374                             }, pr_nextnext
375                         );
376                     }
377                     (void) spmd<>::sync();
378                 }
379                 for( size_t k = 0; k < spmd<>::nprocs(); ++k ) {
380                     if( spmd<>::pid() == k ) {
381                         std::cout « "old pr \t scaled input \t alpha * pr * H at PID "
382                             « k « "\n";
383                         (void) eWiseLambda(
384                             [ &pr, &pr_next, &pr_nextnext ]( const size_t i ) {
385                                 #pragma omp critical
386                                 {
387                                     std::cout « pr[ i ] « "\t" « pr_next[ i ] « "\t"
388                                         « pr_nextnext[ i ] « "\n";
389                                 }
390                             }, pr, pr_next, pr_nextnext
391                         );
392                     }
393                     (void) spmd<>::sync();
394                 }
395 #endif
396
397                 // calculate & normalise new pr and calculate residual
398                 if( ret == SUCCESS ) {
399                     // can we reduce via lambdas?
400                     if( grb::Properties<>::writableCaptured ) {
401                         // yes, we can. So update pr[ i ] and calculate residual simultaneously
402                         ret = eWiseLambda(
403                             [ &pr, &pr_nextnext, &dangling, &residual, &zero ]( const size_t i ) {
404                                 // cache old pagerank vector
405                                 const IOType oldval = pr[ i ];
406                                 // set new pagerank vector
407                                 pr[ i ] = pr_nextnext[ i ] + dangling;
408                                 // update residual
409                                 if( oldval > pr[ i ] ) {
410                                     residual += oldval - pr[ i ];
411                                 } else {
412                                     residual += pr[ i ] - oldval;
413                                 }
414                                 pr_nextnext[ i ] = zero;
415                             }, pr, pr_nextnext
416                         );
417                         // reduce process-local residual
418                         assert( ret == SUCCESS );
419                         if( ret == SUCCESS ) {
420                             ret = grb::collectives<>::allreduce(
421                                 residual,
422                                 grb::operators::add< double >()
423                             );
424                         }
425                         assert( ret == SUCCESS );
426                     } else {
427                         // we cannot reduce via lambdas, so calculate new pr vector
428                         ret = foldl< descriptors::dense >( pr_nextnext, dangling, addM );
429                         assert( ret == SUCCESS );
430                         // do a dot product under the one-norm "ring"
431                         if( ret == SUCCESS ) {
432                             residual = zero;
433                             ret = dot< descriptors::dense >(
434                                 residual,
435                                 pr, pr_nextnext,
436                                 addM, grb::operators::abs_diff< IOType >()
437                             );
438                             assert( ret == SUCCESS );
439                         }
440                         if( ret == SUCCESS ) {
441                             // next pr vector becomes current pr vector
```

```
442                             std::swap( pr, pr_nextnext );
443                         }
444                     }
445                 }
446 
447                 // update iteration count
448                 ++iter;
449 
450                 // check convergence
451                 if( conv != zero && residual <= conv ) { break; }
452 
453 #ifdef _DEBUG
454                 if( grb::spmd<>::pid() == 0 ) {
455                     std::cout « "Iteration " « iter « ", "
456                         « "residual = " « residual « std::endl;
457                 }
458 #endif
459 
460             } while( ret == SUCCESS && iter < max );
461 
462             // check if the user requested any stats, and output if yes
463             if( iterations != nullptr ) {
464                 *iterations = iter;
465             }
466             if( quality != nullptr ) {
467                 *quality = residual;
468             }
469 
470             // return the appropriate exit code
471             if( ret != SUCCESS ) {
472                 if( spmd<>::pid() == 0 ) {
473                     std::cerr « "Error while running simple pagerank algorithm: "
474                         « toString( ret ) « "\n";
475                 }
476                 return ret;
477             } else if( residual <= conv ) {
478 #ifdef _DEBUG
479                 if( spmd<>::pid() == 0 ) {
480                     std::cerr « "Info: simple pagerank converged after " « iter
481                         « " iterations.\n";
482                 }
483 #endif
484                 return SUCCESS; // converged!
485             } else {
486 #ifdef _DEBUG
487                 if( spmd<>::pid() == 0 ) {
488                     std::cout « "Info: simple pagerank did not converge after "
489                         « iter « " iterations.\n";
490                 }
491 #endif
492                 return FAILED; // not converged
493             }
494         }
495 
496     } // namespace algorithms
497 
498 } // namespace grb
499 
500 #endif // end _H_GRB_PAGERANK
501 
```

10.27 sparse_nn_single_inference.hpp File Reference

Implements (non-batched) sparse neural network inference.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<Descriptor descr = descriptors::no_operation, typename IOType , typename WeightType , typename BiasType , typename ThresholdType = IOType, class MinMonoid = Monoid< grb::operators::min< IOType >, grb::identities::infinity >, class ReluMonoid = Monoid< grb::operators::relu< IOType >, grb::identities::negative_infinity >, class Ring = Semiring< grb::operators::add< IOType >, grb::operators::mul< IOType >, grb::identities::zero, grb::identities::one >>
  grb::RC sparse_nn_single_inference (grb::Vector< IOType > &out, const grb::Vector< IOType > &in, const std::vector< grb::Matrix< WeightType > > &layers, const std::vector< BiasType > &biases, const ThresholdType threshold, grb::Vector< IOType > &temp, const ReluMonoid &relu=ReluMonoid(), const Min↩ Monoid &min=MinMonoid(), const Ring &ring=Ring())

  Performs an inference step of a single data element through a Sparse Neural Network defined by num_layers sparse weight matrices and num_layers biases.
- template<Descriptor descr = descriptors::no_operation, typename IOType , typename WeightType , typename BiasType , class Relu↩ Monoid = Monoid< grb::operators::relu< IOType >, grb::identities::negative_infinity >, class Ring = Semiring< grb::operators::add< IOType >, grb::operators::mul< IOType >, grb::identities::zero, grb::identities::one >>
  grb::RC sparse_nn_single_inference (grb::Vector< IOType > &out, const grb::Vector< IOType > &in, const std::vector< grb::Matrix< WeightType > > &layers, const std::vector< BiasType > &biases, grb::Vector< IOType > &temp, const ReluMonoid &relu=ReluMonoid(), const Ring &ring=Ring())

  Performs an inference step of a single data element through a Sparse Neural Network defined by num_layers sparse weight matrices and num_layers biases.

10.27.1 Detailed Description

Implements (non-batched) sparse neural network inference.

Author

Aristeidis Mastoras

10.28 sparse_nn_single_inference.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
26 #ifndef _H_GRB_ALGORITHMS_SPARSE_NN_SINGLE_INFERENCE
27 #define _H_GRB_ALGORITHMS_SPARSE_NN_SINGLE_INFERENCE
28
29 #include <limits>
30 #include <graphblas.hpp>
31
32
33 namespace grb {
34
35     namespace algorithms {
36
37         namespace internal {
38
47             template<
48                 Descriptor descr,
```

```
49                  bool thresholded, typename ThresholdType,
50                  typename IOType, typename WeightType, typename BiasType,
51                  class ReluMonoid, class Ring, class MinMonoid
52              >
53              grb::RC sparse_nn_single_inference(
54                  grb::Vector< IOType > &out,
55                  const grb::Vector< IOType > &in,
56                  const std::vector< grb::Matrix< WeightType > > &layers,
57                  const std::vector< BiasType > &biases,
58                  const ThresholdType threshold,
59                  grb::Vector< IOType > &temp,
60                  const ReluMonoid &relu,
61                  const MinMonoid &min,
62                  const Ring &ring
63              ) {
64                  static_assert( !(descr & descriptors::no_casting) ||
65                      (
66                          std::is_same< IOType, WeightType >::value &&
67                          std::is_same< IOType, BiasType >::value
68                      ), "Input containers have different domains even though the no_casting"
69                      "descriptor was given"
70                  );
71
72                  const size_t num_layers = layers.size();
73
74                  // run-time checks
75                  {
76                      const size_t n = grb::size( out );
77                      if( num_layers == 0 ) {
78                          return ILLEGAL;
79                      }
80                      if( biases.size() != num_layers ) {
81                          return ILLEGAL;
82                      }
83                      if( grb::size( in ) != grb::nrows( ( layers[ 0 ] ) ) ||
84                          grb::size( out ) != grb::ncols( ( layers[ num_layers - 1 ] ) ) ||
85                          grb::size( out ) != grb::size( temp )
86                      ) {
87                          return MISMATCH;
88                      }
89                      for( size_t i = 1; i < num_layers; ++i ) {
90                          if( grb::ncols( ( layers[ i - 1 ] ) ) != grb::nrows( ( layers[ i ] ) ) ) {
91                              return MISMATCH;
92                          }
93                      }
94                      for( size_t i = 0; i < num_layers; ++i ) {
95                          if( grb::ncols( ( layers[ i ] ) ) != grb::nrows( ( layers[ i ] ) ) ) {
96                              return ILLEGAL;
97                          }
98                      }
99                      assert( n == grb::size( in ) );
100                     assert( n == grb::size( temp ) );
101                     if( grb::capacity( out ) != n ) {
102                         return ILLEGAL;
103                     }
104                     if( grb::capacity( temp ) != n ) {
105                         return ILLEGAL;
106                     }
107                 }
108
109                 grb::RC ret = SUCCESS;
110
111     /*
112                 // this is a correct implementation that does not unroll the first and the last
    iterations
113                 // we do not use it because it requires setting the input vector to the output vector
114                 // which results in copying data for 2*n elements
115
116                 ret = grb::set( out, in );
117
118                 for( size_t i = 1; ret == SUCCESS && i < num_layers ; ++i ) {
119
120                     std::swap( out, temp );
121                     ret = ret ? ret : grb::set( out, 0 );
122                     ret = ret ? ret : grb::vxm( out, temp, *(layers[ i - 1 ]), ring );
123                     ret = ret ? ret : grb::foldl< descriptors::dense >( out, biases[ i ],
    ring.getAdditiveMonoid() );
124                     ret = ret ? ret : grb::foldl< descriptors::dense >( out, 0, relu );
125                     if( thresholded ) {
126                         ret = ret ? ret : grb::foldl< descriptors::dense >( out, threshold, min );
127                     }
128                 }
129     */
130
131                 ret = grb::set( out, 0 ); assert( ret == SUCCESS );
132
133                 ret = ret ? ret : grb::vxm( out, in, ( layers[ 0 ] ), ring );
```

```
134                 assert( ret == SUCCESS );
135
136                 ret = ret ? ret : grb::foldl< descriptors::dense >(
137                     out, biases[ 1 ], ring.getAdditiveMonoid()
138                 ); assert( ret == SUCCESS );
139
140                 for( size_t i = 1; ret == SUCCESS && i < num_layers - 1; ++i ) {
141
142                     ret = ret ? ret : grb::foldl< descriptors::dense >( out, 0, relu );
143                     assert( ret == SUCCESS );
144
145                     if( thresholded ) {
146                         ret = ret ? ret : grb::foldl< descriptors::dense >( out, threshold, min );
147                         assert( ret == SUCCESS );
148                     }
149
150                     if( ret == SUCCESS ) {
151                         std::swap( out, temp );
152                     }
153
154                     ret = grb::set( out, 0 );
155                     assert( ret == SUCCESS );
156
157                     ret = ret ? ret : grb::vxm< descriptors::dense >(
158                         out, temp, ( layers[ i ] ), ring
159                     ); assert( ret == SUCCESS );
160
161                     ret = ret ? ret : grb::foldl< descriptors::dense >(
162                         out, biases[ i + 1 ], ring.getAdditiveMonoid()
163                     ); assert( ret == SUCCESS );
164                 }
165
166                 ret = ret ? ret : grb::foldl< descriptors::dense >( out, 0, relu );
167                 assert( ret == SUCCESS );
168
169                 if( thresholded ) {
170                     ret = ret ? ret : grb::foldl< descriptors::dense >( out, threshold, min );
171                     assert( ret == SUCCESS );
172                 }
173
174                 return ret;
175             }
176
177         } // end namespace "grb::internal"
178
257         template< Descriptor descr = descriptors::no_operation,
258             typename IOType,
259             typename WeightType,
260             typename BiasType,
261             class ReluMonoid = Monoid<
262                 grb::operators::relu< IOType >,
263                 grb::identities::negative_infinity
264             >,
265             class Ring = Semiring<
266                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
267                 grb::identities::zero, grb::identities::one
268             >
269         >
270         grb::RC sparse_nn_single_inference(
271             grb::Vector< IOType > &out,
272             const grb::Vector< IOType > &in,
273             const std::vector< grb::Matrix< WeightType > > &layers,
274             const std::vector< BiasType > &biases,
275             grb::Vector< IOType > &temp,
276             const ReluMonoid &relu = ReluMonoid(),
277             const Ring &ring = Ring()
278         ) {
279             static_assert( !(descr & descriptors::no_casting) ||
280                 (
281                     std::is_same< IOType, WeightType >::value &&
282                     std::is_same< IOType, BiasType >::value
283                 ), "Input containers have different domains even though the no_casting "
284                 "descriptor was given" );
285             Monoid<
286                 grb::operators::min< IOType >, grb::identities::infinity
287             > dummyTresholdMonoid;
288             return internal::sparse_nn_single_inference<
289                 descr, false, double
290             > (
291                 out, in, layers,
292                 biases, 0.0,
293                 temp,
294                 relu, dummyTresholdMonoid, ring
295             );
296         }
297
383         template< Descriptor descr = descriptors::no_operation,
```

```
384             typename IOType,
385             typename WeightType,
386             typename BiasType,
387             typename ThresholdType = IOType,
388             class MinMonoid = Monoid<
389                 grb::operators::min< IOType >, grb::identities::infinity
390             >,
391             class ReluMonoid = Monoid<
392                 grb::operators::relu< IOType >,
393                 grb::identities::negative_infinity
394             >,
395             class Ring = Semiring<
396                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
397                 grb::identities::zero, grb::identities::one
398             >
399         >
400         grb::RC sparse_nn_single_inference(
401             grb::Vector< IOType > &out,
402             const grb::Vector< IOType > &in,
403             const std::vector< grb::Matrix< WeightType > > &layers,
404             const std::vector< BiasType > &biases,
405             const ThresholdType threshold,
406             grb::Vector< IOType > &temp,
407             const ReluMonoid &relu = ReluMonoid(),
408             const MinMonoid &min = MinMonoid(),
409             const Ring &ring = Ring()
410         ) {
411             static_assert( !(descr & descriptors::no_casting) ||
412                 (
413                     std::is_same< IOType, WeightType >::value &&
414                     std::is_same< IOType, BiasType >::value
415                 ), "Input containers have different domains even though the no_casting "
416                 "descriptor was given" );
417             return internal::sparse_nn_single_inference<
418                 descr, true
419             > (
420                 out, in, layers,
421                 biases, threshold,
422                 temp,
423                 relu, min, ring
424             );
425         }
426
427     } // namespace algorithms
428
429 } // end namespace grb
430
431 #endif // end _H_GRB_ALGORITHMS_SPARSE_NN_SINGLE_INFERENCE
432
```

10.29 spy.hpp File Reference

Implements a simple matrix spy algorithm.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::algorithms

  The namespace for ALP/GraphBLAS algorithms.

Functions

- template<bool normalize = false, typename IOType >
  RC spy (grb::Matrix< IOType > &out, const grb::Matrix< bool > &in)

  Specialisation for boolean input matrices in.
- template<bool normalize = false, typename IOType , typename InputType >
  RC spy (grb::Matrix< IOType > &out, const grb::Matrix< InputType > &in)

Given an input matrix and a smaller output matrix, map nonzeroes from the input matrix into the smaller one and count the number of nonzeroes that are mapped from the bigger matrix into the smaller.

- template<bool normalize = false, typename IOType >
  RC spy (grb::Matrix< IOType > &out, const grb::Matrix< void > &in)

  Specialisation for void input matrices in.

10.29.1 Detailed Description

Implements a simple matrix spy algorithm.

Author

A. N. Yzelman

10.30 spy.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10 *
11 * Unless required by applicable law or agreed to in writing, software
12 * distributed under the License is distributed on an "AS IS" BASIS,
13 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 * See the License for the specific language governing permissions and
15 * limitations under the License.
16 */
17
26 #ifndef _H_GRB_ALGORITHMS_SPY
27 #define _H_GRB_ALGORITHMS_SPY
28
29 #include <type_traits>
30 #include <vector>
31
32 #include <graphblas.hpp>
33
34
35 namespace grb {
36
37     namespace algorithms {
38
39         namespace internal {
40
47             template< bool normalize, typename IOType, typename InputType >
48             RC spy_from_bool_or_void_input(
49                 grb::Matrix< IOType > &out, const grb::Matrix< InputType > &in,
50                 const size_t m, const size_t n,
51                 const size_t small_m, const size_t small_n
52             ) {
53                 static_assert( std::is_same< InputType, bool >::value ||
54                     std::is_same< InputType, void >::value,
55                     "Error in call to internal::spy_from_bool_or_void_input"
56                 );
57
58                 // Q must be n by small_n
59                 grb::Matrix< unsigned char > Q( n, small_n );
60                 grb::RC ret = grb::resize( Q, n );
61                 // TODO FIXME use repeating + auto-incrementing iterators
62                 std::vector< size_t > I, J;
63                 std::vector< unsigned char > V;
64                 const double n_sample = static_cast< double >(n) /
65                     static_cast< double >(small_n);
66                 for( size_t i = 0; i < n; ++i ) {
67                     I.push_back( i );
68                     J.push_back( static_cast< double >(i) / n_sample );
69                     V.push_back( 1 );
```

```
70                  }
71                  ret = grb::buildMatrixUnique(
72                      Q, &(I[0]), &(J[0]), &(V[0]), n,
73                      grb::SEQUENTIAL
74                  );
75
76                  // P must be small_m by m
77                  grb::Matrix< unsigned char > P( small_m, m );
78                  if( ret == SUCCESS ) {
79                      ret = grb::resize( P, m );
80                      // TODO FIXME use repeating + auto-incrementing iterators
81                      std::vector< size_t > I, J;
82                      std::vector< unsigned char > V;
83                      const double m_sample = static_cast< double >(m) /
84                          static_cast< double >(small_m);
85                      for( size_t i = 0; i < m; ++i ) {
86                          I.push_back( static_cast< double >(i) / m_sample );
87                          J.push_back( i );
88                          V.push_back( 1 );
89                      }
90                      ret = grb::buildMatrixUnique(
91                          P, &(I[0]), &(J[0]), &(V[0]), m,
92                          grb::SEQUENTIAL
93                      );
94                  }
95
96                  // tmp must be m by small_n OR small_m by n
97                  if( ret == SUCCESS && m - small_m > n - small_n ) {
98                      grb::Semiring<
99                          grb::operators::add< size_t >,
100                          grb::operators::left_assign_if< size_t, bool, size_t >,
101                          grb::identities::zero,
102                          grb::identities::logical_true
103                      > leftAssignAndAdd;
104                      grb::Matrix< size_t > tmp( small_m, n );
105                      ret = ret ? ret : grb::mxm( tmp, P, in, leftAssignAndAdd, RESIZE );
106                      ret = ret ? ret : grb::mxm( tmp, P, in, leftAssignAndAdd, EXECUTE );
107                      ret = ret ? ret : grb::mxm( out, tmp, Q, leftAssignAndAdd, RESIZE );
108                      ret = ret ? ret : grb::mxm( out, tmp, Q, leftAssignAndAdd, EXECUTE );
109                  } else {
110                      grb::Semiring<
111                          grb::operators::add< size_t >,
112                          grb::operators::right_assign_if< bool, size_t, size_t >,
113                          grb::identities::zero,
114                          grb::identities::logical_true
115                      > rightAssignAndAdd;
116                      grb::Matrix< size_t > tmp( m, small_n );
117                      ret = ret ? ret : grb::mxm( tmp, in, Q, rightAssignAndAdd, RESIZE );
118                      ret = ret ? ret : grb::mxm( tmp, in, Q, rightAssignAndAdd, EXECUTE );
119                      ret = ret ? ret : grb::mxm( out, P, tmp, rightAssignAndAdd, RESIZE );
120                      ret = ret ? ret : grb::mxm( out, P, tmp, rightAssignAndAdd, EXECUTE );
121                  }
122
123                  if( ret == SUCCESS && normalize ) {
124                      ret = grb::eWiseLambda( [] (const size_t, const size_t, IOType &v ) {
125                          assert( v > 0 );
126                          v = static_cast< IOType >( 1 ) / v;
127                      }, out );
128                  }
129
130                  return ret;
131              }
132
133          }
134
194          template<
195              bool normalize = false,
196              typename IOType, typename InputType
197          >
198          RC spy( grb::Matrix< IOType > &out, const grb::Matrix< InputType > &in ) {
199              static_assert( !normalize || std::is_floating_point< IOType >::value,
200                  "When requesting a normalised spy plot, the data type must be "
201                  "floating-point"
202              );
203
204              const size_t m = grb::nrows( in );
205              const size_t n = grb::ncols( in );
206              const size_t small_m = grb::nrows( out );
207              const size_t small_n = grb::ncols( out );
208
209              // runtime checks and shortcuts
210              if( small_m > m ) { return ILLEGAL; }
211              if( small_n > n ) { return ILLEGAL; }
212              if( small_m == m && small_n == n ) {
213                  return grb::set< grb::descriptors::structural >( out, in, 1 );
214              }
215
```

```
216             grb::RC ret = grb::clear( out );
217
218             grb::Matrix< bool > tmp( m, n );
219             ret = ret ? ret : grb::resize( tmp, grb::nnz( in ) );
220             ret = ret ? ret : grb::set< grb::descriptors::structural >( tmp, in, true );
221             ret = ret ? ret : grb::algorithms::internal::template
222                 spy_from_bool_or_void_input< normalize >(
223                     out, tmp, m, n, small_m, small_n
224                 );
225
226             return ret;
227         }
228
230         template< bool normalize = false, typename IOType >
231         RC spy( grb::Matrix< IOType > &out, const grb::Matrix< bool > &in ) {
232             static_assert( !normalize || std::is_floating_point< IOType >::value,
233                 "When requesting a normalised spy plot, the data type must be "
234                 "floating-point" );
235
236             const size_t m = grb::nrows( in );
237             const size_t n = grb::ncols( in );
238             const size_t small_m = grb::nrows( out );
239             const size_t small_n = grb::ncols( out );
240
241             // runtime checks and shortcuts
242             if( small_m > m ) { return ILLEGAL; }
243             if( small_n > n ) { return ILLEGAL; }
244             if( small_m == m && small_n == n ) {
245                 return grb::set< grb::descriptors::structural >( out, in, 1 );
246             }
247
248             grb::RC ret = grb::clear( out );
249
250             ret = ret ? ret : grb::algorithms::internal::template
251                 spy_from_bool_or_void_input< normalize >(
252                     out, in, m, n, small_m, small_n
253                 );
254
255             return ret;
256         }
257
259         template< bool normalize = false, typename IOType >
260         RC spy( grb::Matrix< IOType > &out, const grb::Matrix< void > &in ) {
261             static_assert( !normalize || std::is_floating_point< IOType >::value,
262                 "When requesting a normalised spy plot, the data type must be "
263                 "floating-point"
264             );
265
266             const size_t m = grb::nrows( in );
267             const size_t n = grb::ncols( in );
268             const size_t small_m = grb::nrows( out );
269             const size_t small_n = grb::ncols( out );
270
271             // runtime checks and shortcuts
272             if( small_m > m ) { return ILLEGAL; }
273             if( small_n > n ) { return ILLEGAL; }
274             if( small_m == m && small_n == n ) {
275                 return grb::set< grb::descriptors::structural >( out, in, 1 );
276             }
277
278             grb::RC ret = grb::clear( out );
279
280             ret = ret ? ret : grb::algorithms::internal::template
281                 spy_from_bool_or_void_input< normalize >(
282                     out, in, m, n, small_m, small_n
283                 );
284
285             return ret;
286         }
287
288     } // end namespace "grb::algorithms"
289
290 } // end namespace "grb"
291
292 #endif // _H_GRB_ALGORITHMS_SPY
293
```

10.31 backends.hpp File Reference

This file contains a register of all backends that are either implemented, under implementation, or conceived and recorded for future consideration to implement.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Enumerations

- enum Backend {
  reference , reference_omp , hyperdags , nonblocking ,
  shmem1D , **NUMA1D** , **GENERIC_BSP** , BSP1D ,
  doublyBSP1D , **BSP2D** , **autoBSP** , **optBSP** ,
  hybrid , **hybridSmall** , **hybridMid** , **hybridLarge** ,
  minFootprint , banshee , **banshee_ssr** }

  A collection of all backends.

10.31.1 Detailed Description

This file contains a register of all backends that are either implemented, under implementation, or conceived and recorded for future consideration to implement.

Author

: A. N. Yzelman

Date

21st of December, 2016

10.32 backends.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
29 #ifndef _H_GRB_BACKENDS
30 #define _H_GRB_BACKENDS
31
32
33 namespace grb {
34
46     enum Backend {
47
52         reference,
53
59         reference_omp,
60
67         hyperdags,
68
```

```
73          nonblocking,
74
81          shmem1D,
82
90          NUMA1D,
91
98          GENERIC_BSP,
99
110          BSP1D,
111
120          doublyBSP1D,
121
130          BSP2D,
131
142          autoBSP,
143
151          optBSP,
152
160          hybrid,
161
169          hybridSmall,
170
180          hybridMid,
181
194          hybridLarge,
195
201          minFootprint,
202
207          banshee,
208
217          banshee_ssr
218
219      };
220
221 } // namespace grb
222
223 #endif
224
```

10.33 benchmark.hpp File Reference

This file contains a variant on the grb::Launcher specialised for benchmarks.

Classes

- class Benchmarker< mode, implementation >

  A class that follows the API of the grb::Launcher, but instead of launching the given ALP program once, it launches it multiple times while benchmarking its execution times.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.33.1 Detailed Description

This file contains a variant on the grb::Launcher specialised for benchmarks.

Author

J. W. Nash & A. N. Yzelman

Date

17th of April, 2017

10.34 benchmark.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
28 #ifndef _H_GRB_BENCH_BASE
29 #define _H_GRB_BENCH_BASE
30
31 #include <chrono>
32 #include <ios>
33 #include <limits>
34 #include <string>
35
36 #include <graphblas/backends.hpp>
37 #include <graphblas/ops.hpp>
38 #include <graphblas/rc.hpp>
39 #include <graphblas/utils.hpp>
40 #include <graphblas/utils/TimerResults.hpp>
41
42 #include "collectives.hpp"
43 #include "config.hpp"
44 #include "exec.hpp"
45
46 #ifndef _GRB_NO_STDIO
47  #include <iostream>
48 #endif
49
50 #ifndef _GRB_NO_EXCEPTIONS
51  #include <stdexcept>
52 #endif
53
54 #include <math.h>
55
56
82 namespace grb {
83
84     namespace internal {
85
91         class BenchmarkerBase {
92
93             protected:
94
95 #ifndef _GRB_NO_STDIO
104                 static void printTimeSinceEpoch( const bool printHeader = true ) {
105                     const auto now = std::chrono::system_clock::now();
106                     const auto since = now.time_since_epoch();
107                     if( printHeader ) {
108                         std::cout « "Time since epoch (in ms.): ";
109                     }
110                     std::cout « std::chrono::duration_cast<
111                             std::chrono::milliseconds
112                         >( since ).count() « "\n";
113                 }
114 #endif
115
119                 static void benchmark_calc_inner(
120                     const size_t loop,
121                     const size_t total,
122                     grb::utils::TimerResults &inner_times,
123                     grb::utils::TimerResults &total_times,
124                     grb::utils::TimerResults &min_times,
125                     grb::utils::TimerResults &max_times,
126                     grb::utils::TimerResults * sdev_times
127                 ) {
128                     inner_times.normalize( total );
129                     total_times.accum( inner_times );
130                     min_times.min( inner_times );
131                     max_times.max( inner_times );
132                     sdev_times[ loop ] = inner_times;
133                 }
```

```
134
138                 static void benchmark_calc_outer(
139                     const size_t total,
140                     grb::utils::TimerResults &total_times,
141                     grb::utils::TimerResults &min_times,
142                     grb::utils::TimerResults &max_times,
143                     grb::utils::TimerResults * sdev_times,
144                     const size_t pid
145                 ) {
146                     total_times.normalize( total );
147                     grb::utils::TimerResults sdev;
148                     // compute standard dev of average times, leaving sqrt calculation until
149                     // the output of the values
150                     sdev.set( 0 );
151                     for( size_t i = 0; i < total; i++ ) {
152                         double diff = sdev_times[ i ].io - total_times.io;
153                         sdev.io += diff * diff;
154                         diff = sdev_times[ i ].preamble - total_times.preamble;
155                         sdev.preamble += diff * diff;
156                         diff = sdev_times[ i ].useful - total_times.useful;
157                         sdev.useful += diff * diff;
158                         diff = sdev_times[ i ].postamble - total_times.postamble;
159                         sdev.postamble += diff * diff;
160                     }
161                     // unbiased normalisation of the standard deviation
162                     sdev.normalize( total - 1 );
163
164 #ifndef _GRB_NO_STDIO
165                     // output results
166                     if( pid == 0 ) {
167                         std::cout « "Overall timings (io, preamble, useful, postamble):\n"
168                             « std::scientific;
169                         std::cout « "Avg: " « total_times.io « ", " « total_times.preamble
170                             « ", " « total_times.useful « ", " « total_times.postamble « "\n";
171                         std::cout « "Min: " « min_times.io « ", " « min_times.preamble « ", "
172                             « min_times.useful « ", " « min_times.postamble « "\n";
173                         std::cout « "Max: " « max_times.io « ", " « max_times.preamble « ", "
174                             « max_times.useful « ", " « max_times.postamble « "\n";
175                         std::cout « "Std: " « sqrt( sdev.io ) « ", " « sqrt( sdev.preamble )
176                             « ", " « sqrt( sdev.useful ) « ", " « sqrt( sdev.postamble ) « "\n";
177  #if __GNUC__ > 4
178                         std::cout « std::defaultfloat;
179  #endif
180                         printTimeSinceEpoch();
181                     }
182 #else
183                     // we ran the benchmark, but may not have a way to output it in this case
184                     // this currently only is touched by the #grb::banshee backend, which
185                     // provides other timing mechanisms.
186                     (void) min_times;
187                     (void) max_times;
188                     (void) pid;
189 #endif
190                 }
191
213                 template<
214                     typename U,
215                     enum Backend implementation = config::default_backend
216                 >
217                 static RC benchmark(
218                     void ( *alp_program )( const void *, const size_t, U & ),
219                     const void * data_in,
220                     const size_t in_size,
221                     U &data_out,
222                     const size_t inner,
223                     const size_t outer,
224                     const size_t pid
225                 ) {
226                     const double inf = std::numeric_limits< double >::infinity();
227                     grb::utils::TimerResults total_times, min_times, max_times;
228                     grb::utils::TimerResults * sdev_times =
229                         new grb::utils::TimerResults[ outer ];
230                     total_times.set( 0 );
231                     min_times.set( inf );
232                     max_times.set( 0 );
233
234                     // outer loop
235                     for( size_t out = 0; out < outer; ++out ) {
236                         grb::utils::TimerResults inner_times;
237                         inner_times.set( 0 );
238
239                         // inner loop
240                         for( size_t in = 0; in < inner; in++ ) {
241                             data_out.times.set( 0 );
242                             ( *alp_program )( data_in, in_size, data_out );
243                             grb::collectives< implementation >::reduce(
244                                 data_out.times.io, 0, grb::operators::max< double >() );
```

```
245                             grb::collectives< implementation >::reduce(
246                                 data_out.times.preamble, 0, grb::operators::max< double >() );
247                             grb::collectives< implementation >::reduce(
248                                 data_out.times.useful, 0, grb::operators::max< double >() );
249                             grb::collectives< implementation >::reduce(
250                                 data_out.times.postamble, 0, grb::operators::max< double >() );
251                             inner_times.accum( data_out.times );
252                         }
253 
254                         // calculate performance stats
255                         benchmark_calc_inner( out, inner, inner_times, total_times, min_times,
256                             max_times, sdev_times );
257 
258 #ifndef _GRB_NO_STDIO
259                         // give experiment output line
260                         if( pid == 0 ) {
261                             std::cout << "Outer iteration #" << out << " timings (io, preamble, "
262                                 << "useful, postamble, time since epoch): ";
263                             std::cout << inner_times.io << ", " << inner_times.preamble << ", "
264                                 << inner_times.useful << ", " << inner_times.postamble << ", ";
265                             printTimeSinceEpoch( false );
266                         }
267 #endif
268 
269                         // pause for next outer loop
270                         if( sleep( 1 ) != 0 ) {
271 #ifndef _GRB_NO_STDIO
272                             std::cerr << "Sleep interrupted, assume benchmark is unreliable; "
273                                 << "exiting.\n";
274 #endif
275                             abort();
276                         }
277                     }
278 
279                     // calculate performance stats
280                     benchmark_calc_outer( outer, total_times, min_times, max_times, sdev_times,
281                         pid );
282                     delete [] sdev_times;
283 
284                     return SUCCESS;
285                 }
286 
308                 template<
309                     typename T, typename U,
310                     enum Backend implementation = config::default_backend
311                 >
312                 static RC benchmark(
313                     void ( *alp_program )( const T &, U & ),
314                     const T &data_in,
315                     U &data_out,
316                     const size_t inner,
317                     const size_t outer,
318                     const size_t pid
319                 ) {
320                     const double inf = std::numeric_limits< double >::infinity();
321                     grb::utils::TimerResults total_times, min_times, max_times;
322                     grb::utils::TimerResults * sdev_times =
323                         new grb::utils::TimerResults[ outer ];
324                     total_times.set( 0 );
325                     min_times.set( inf );
326                     max_times.set( 0 );
327 
328                     // outer loop
329                     for( size_t out = 0; out < outer; ++out ) {
330                         grb::utils::TimerResults inner_times;
331                         inner_times.set( 0 );
332 
333                         // inner loop
334                         for( size_t in = 0; in < inner; ++in ) {
335                             data_out.times.set( 0 );
336 
337                             ( *alp_program )( data_in, data_out );
338                             grb::collectives< implementation >::reduce( data_out.times.io, 0,
339                                 grb::operators::max< double >() );
340                             grb::collectives< implementation >::reduce( data_out.times.preamble, 0,
341                                 grb::operators::max< double >() );
342                             grb::collectives< implementation >::reduce( data_out.times.useful, 0,
343                                 grb::operators::max< double >() );
344                             grb::collectives< implementation >::reduce( data_out.times.postamble, 0,
345                                 grb::operators::max< double >() );
346                             inner_times.accum( data_out.times );
347                         }
348 
349                         // calculate performance stats
350                         benchmark_calc_inner( out, inner, inner_times, total_times, min_times,
351                             max_times, sdev_times );
352 
```

```
353 #ifndef _GRB_NO_STDIO
354                         // give experiment output line
355                         if( pid == 0 ) {
356                             std::cout « "Outer iteration #" « out « " timings "
357                                 « "(io, preamble, useful, postamble, time since epoch): " « std::fixed
358                                 « inner_times.io « ", " « inner_times.preamble « ", "
359                                 « inner_times.useful « ", " « inner_times.postamble « ", ";
360                                 printTimeSinceEpoch( false );
361                             std::cout « std::scientific;
362                         }
363 #endif
364 
365                         // pause for next outer loop
366                         if( sleep( 1 ) != 0 ) {
367 #ifndef _GRB_NO_STDIO
368                             std::cerr « "Sleep interrupted, assume benchmark is unreliable; "
369                                 « "exiting.\n";
370 #endif
371                             abort();
372                         }
373                     }
374 
375                     // calculate performance stats
376                     benchmark_calc_outer( outer, total_times, min_times, max_times, sdev_times,
377                         pid );
378                     delete[] sdev_times;
379 
380                     return SUCCESS;
381                 }
382 
383 
384             public:
385 
386                 BenchmarkerBase() {
387 #ifndef _GRB_NO_STDIO
388                     printTimeSinceEpoch();
389 #endif
390                 }
391 
392         };
393 
394     } // namespace internal
395 
404     template< enum EXEC_MODE mode, enum Backend implementation >
405     class Benchmarker {
406 
407         public :
408 
437             Benchmarker(
438                 const size_t process_id = 0,
439                 size_t nprocs = 1,
440                 std::string hostname = "localhost",
441                 std::string port = "0"
442             ) {
443                 (void)process_id; (void)nprocs; (void)hostname; (void)port;
444 #ifndef _GRB_NO_EXCEPTIONS
445                 throw std::logic_error( "Benchmarker class called with unsupported mode or "
446                     "implementation" );
447 #endif
448             }
449 
485             template< typename T, typename U >
486             RC exec(
487                 void ( *alp_program )( const T &, U & ),
488                 const T &data_in,
489                 U &data_out,
490                 const size_t inner,
491                 const size_t outer,
492                 const bool broadcast = false
493             ) const {
494                 (void) alp_program;
495                 (void) data_in;
496                 (void) data_out;
497                 (void) inner;
498                 (void) outer;
499                 (void) broadcast;
500 
501                 // stub implementation, should be overridden by specialised implementation.
502                 // furthermore, it should be impossible to call this function without
503                 // triggering an exception during construction of this stub class, so we
504                 // just return PANIC here
505                 return PANIC;
506             }
507 
545             template< typename U >
546             RC exec(
547                 void ( *alp_program )( const void *, const size_t, U & ),
```

```
548                 const void * data_in, const size_t in_size,
549                 U &data_out,
550                 const size_t inner, const size_t outer,
551                 const bool broadcast = false
552             ) const {
553                 (void) alp_program;
554                 (void) data_in;
555                 (void) in_size;
556                 (void) data_out;
557                 (void) inner;
558                 (void) outer;
559                 (void) broadcast;
560 
561                 // stub implementation, should be overridden by specialised implementation.
562                 // furthermore, it should be impossible to call this function without
563                 // triggering an exception during construction of this stub class, so we
564                 // just return PANIC here
565                 return PANIC;
566             }
567 
591             static RC finalize() {
592                 return Launcher< mode, implementation >::finalize();
593             }
594 
595     };
596 
597 } // end namespace "grb"
598 
599 #endif // end _H_GRB_BENCH_BASE
600 
```

10.35 blas1.hpp File Reference

Defines the ALP/GraphBLAS level-1 API.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Macros

- #define NO_MASK Vector< bool >(0)

  A standard vector to use for mask parameters.

Functions

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , Backend backend, typename Coords >
  RC dot (IOType &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_↩ if< !grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< IOType >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the dot product, $z+ = (x, y)$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class AddMonoid , class AnyOp , typename OutputType , typename InputType1 , typename InputType2 , enum Backend backend, typename Coords >
  RC dot (OutputType &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const AddMonoid &addMonoid=AddMonoid(), const AnyOp &anyOp=AnyOp(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Add↩ Monoid >::value &&grb::is_operator< AnyOp >::value, void >::type *const =nullptr)

Calculates the dot product, $z+ = (x, y)$, under a given additive monoid and multiplicative operator.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + \beta$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + y$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = x. + \beta$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition of two vectors, $z+ = x. + y$, under a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + \beta$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = \alpha. + y$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  Calculates the element-wise addition, $z+ = x. + \beta$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >

RC eWiseAdd (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

Calculates the element-wise addition of two vectors, $z+ = x. + y$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot \beta$, out of place, monoid version.
- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot \beta$, out of place, operator version.
- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot y$, out of place, monoid version.
- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot y$, out of place, operator version.
- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = x \odot \beta$, out of place, monoid variant.
- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = x \odot \beta$, out of place, operator variant.
- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Computes $z = x \odot y$, out of place, monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = x \odot y$, out of place, operator variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot \beta$, out of place, masked monoid version.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot \beta$, out of place, operator and masked version.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot y$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = \alpha \odot y$, out of place, masked operator version.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = x \odot \beta$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

Computes $z = x \odot \beta$, out of place, masked operator variant.

- template<Descriptor descr = descriptors::no_operation, class Monoid , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Monoid &monoid=Monoid(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Computes $z = x \odot y$, out of place, masked monoid variant.

- template<Descriptor descr = descriptors::no_operation, class OP , enum Backend backend, typename OutputType , typename MaskType , typename InputType1 , typename InputType2 , typename Coords >
  RC eWiseApply (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const OP &op=OP(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Computes $z = x \odot y$, out of place, masked operator variant.

- template<typename Func , typename DataType , Backend backend, typename Coords , typename... Args>
  RC eWiseLambda (const Func f, const Vector< DataType, backend, Coords > &x, Args...)

  Executes an arbitrary element-wise user-defined function f on any number of vectors of equal length.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of two scalars, $z+ = \alpha . * \beta$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a scalar and vector, $z+ = \alpha . * y$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a vector and scalar, $z+ = x . * \beta$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of two vectors, $z+ = x . * y$, under a given semiring.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value

&&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

In-place element-wise multiplication of two scalars, $z+ = \alpha . * \beta$, under a given semiring, masked variant.

- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const InputType1 alpha, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a scalar and vector, $z+ = \alpha . * y$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const InputType2 beta, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of a vector and scalar, $z+ = x . * \beta$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Ring , enum Backend backend, typename InputType1 , typename InputType2 , typename OutputType , typename MaskType , typename Coords >
  RC eWiseMul (Vector< OutputType, backend, Coords > &z, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Ring &ring=Ring(), const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&grb::is_semiring< Ring >::value, void >::type *const =nullptr)

  In-place element-wise multiplication of two vectors, $z+ = x . * y$, under a given semiring, masked variant.
- template<Descriptor descr = descriptors::no_operation, class Monoid , typename IOType , typename InputType , Backend backend, typename Coords >
  RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Folds a vector into a scalar, left-to-right.
- template<Descriptor descr = descriptors::no_operation, class Monoid , typename InputType , typename IOType , typename MaskType , Backend backend, typename Coords >
  RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Vector< MaskType, backend, Coords > &mask, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< InputType >::value &&!grb::is_object< MaskType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Reduces, or folds, a vector into a scalar.
- template<Descriptor descr = descriptors::no_operation, class OP , typename IOType , typename InputType , typename MaskType , Backend backend, typename Coords >
  RC foldl (IOType &x, const Vector< InputType, backend, Coords > &y, const Vector< MaskType, backend, Coords > &mask, const OP &op=OP(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< MaskType >::value &&grb::is_operator< OP >::value, void >::type *const =nullptr)

  Folds a vector into a scalar, left-to-right.
- template<Descriptor descr = descriptors::no_operation, class Monoid , typename InputType , typename IOType , typename MaskType , Backend backend, typename Coords >
  RC foldr (const Vector< InputType, backend, Coords > &x, const Vector< MaskType, backend, Coords > &mask, IOType &y, const Monoid &monoid=Monoid(), const typename std::enable_if< !grb::is_object< IOType >::value &&!grb::is_object< InputType >::value &&!grb::is_object< MaskType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

  Folds a vector into a scalar, right-to-left.
- template<Descriptor descr = descriptors::no_operation, class Monoid , typename IOType , typename InputType , Backend backend, typename Coords >
  RC foldr (const Vector< InputType, backend, Coords > &y, IOType &x, const Monoid &monoid=Monoid(),

const typename std::enable_if< !grb::is_object< IOType >::value &&grb::is_monoid< Monoid >::value, void >::type *const =nullptr)

Folds a vector into a scalar, right-to-left.

10.35.1 Detailed Description

Defines the ALP/GraphBLAS level-1 API.

Author

A. N. Yzelman

Date

5th of December 2016

10.36 blas1.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_BASE_BLAS1
28 #define _H_GRB_BASE_BLAS1
29
30 #include <graphblas/rc.hpp>
31 #include <graphblas/ops.hpp>
32 #include <graphblas/phase.hpp>
33 #include <graphblas/monoid.hpp>
34 #include <graphblas/backends.hpp>
35 #include <graphblas/semiring.hpp>
36 #include <graphblas/descriptors.hpp>
37 #include <graphblas/internalops.hpp>
38
39 #include <assert.h>
40
41
42 namespace grb {
43
154     #define NO_MASK Vector< bool >( 0 )
155
201     template<
202         Descriptor descr = descriptors::no_operation,
203         class OP, enum Backend backend,
204         typename OutputType,
205         typename InputType1, typename InputType2,
206         typename Coords
207     >
208     RC eWiseApply(
209         Vector< OutputType, backend, Coords > &z,
210         const InputType1 alpha,
211         const InputType2 beta,
212         const OP &op = OP(),
213         const Phase &phase = EXECUTE,
214         const typename std::enable_if<
215             !grb::is_object< OutputType >::value &&
```

```
216             !grb::is_object< InputType1 >::value &&
217             !grb::is_object< InputType2 >::value &&
218             grb::is_operator< OP >::value, void
219         >::type * const = nullptr
220     ) {
221 #ifdef _DEBUG
222         std::cout << "In eWiseApply ([T1]<-T2<-T3), operator, base\n";
223 #endif
224 #ifndef NDEBUG
225         const bool should_not_call_eWiseApplyOpASS_base = false;
226         assert( should_not_call_eWiseApplyOpASS_base );
227 #endif
228         (void) z;
229         (void) alpha;
230         (void) beta;
231         (void) op;
232         (void) phase;
233         return UNSUPPORTED;
234     }
235
284     template<
285         Descriptor descr = descriptors::no_operation,
286         class OP, enum Backend backend,
287         typename OutputType, typename MaskType,
288         typename InputType1, typename InputType2,
289         typename Coords
290     >
291     RC eWiseApply(
292         Vector< OutputType, backend, Coords > &z,
293         const Vector< MaskType, backend, Coords > &mask,
294         const InputType1 alpha,
295         const InputType2 beta,
296         const OP &op = OP(),
297         const Phase &phase = EXECUTE,
298         const typename std::enable_if<
299             !grb::is_object< OutputType >::value &&
300             !grb::is_object< MaskType >::value &&
301             !grb::is_object< InputType1 >::value &&
302             !grb::is_object< InputType2 >::value &&
303             grb::is_operator< OP >::value, void
304         >::type * const = nullptr
305     ) {
306 #ifdef _DEBUG
307         std::cout << "In masked eWiseApply ([T1]<-T2<-T3), operator, base\n";
308 #endif
309 #ifndef NDEBUG
310         const bool should_not_call_eWiseApplyOpAMSS_base = false;
311         assert( should_not_call_eWiseApplyOpAMSS_base );
312 #endif
313         (void) z;
314         (void) mask;
315         (void) alpha;
316         (void) beta;
317         (void) op;
318         (void) phase;
319         return UNSUPPORTED;
320     }
321
367     template<
368         Descriptor descr = descriptors::no_operation,
369         class Monoid, enum Backend backend,
370         typename OutputType,
371         typename InputType1, typename InputType2,
372         typename Coords
373     >
374     RC eWiseApply(
375         Vector< OutputType, backend, Coords > &z,
376         const InputType1 alpha,
377         const InputType2 beta,
378         const Monoid &monoid = Monoid(),
379         const Phase &phase = EXECUTE,
380         const typename std::enable_if<
381             !grb::is_object< OutputType >::value &&
382             !grb::is_object< InputType1 >::value &&
383             !grb::is_object< InputType2 >::value &&
384             grb::is_monoid< Monoid >::value, void
385         >::type * const = nullptr
386     ) {
387 #ifdef _DEBUG
388         std::cout << "In eWiseApply ([T1]<-T2<-T3), monoid, base\n";
389 #endif
390 #ifndef NDEBUG
391         const bool should_not_call_eWiseApplyMonASS_base = false;
392         assert( should_not_call_eWiseApplyMonASS_base );
393 #endif
394         (void) z;
395         (void) alpha;
```

```
396         (void) beta;
397         (void) monoid;
398         (void) phase;
399         return UNSUPPORTED;
400     }
401 
450     template<
451         Descriptor descr = descriptors::no_operation,
452         class Monoid, enum Backend backend,
453         typename OutputType, typename MaskType,
454         typename InputType1, typename InputType2,
455         typename Coords
456     >
457     RC eWiseApply(
458         Vector< OutputType, backend, Coords > &z,
459         const Vector< MaskType, backend, Coords > &mask,
460         const InputType1 alpha,
461         const InputType2 beta,
462         const Monoid &monoid = Monoid(),
463         const Phase &phase = EXECUTE,
464         const typename std::enable_if<
465             !grb::is_object< OutputType >::value &&
466             !grb::is_object< MaskType >::value &&
467             !grb::is_object< InputType1 >::value &&
468             !grb::is_object< InputType2 >::value &&
469             grb::is_monoid< Monoid >::value, void
470         >::type * const = nullptr
471     ) {
472 #ifdef _DEBUG
473         std::cout << "In masked eWiseApply ([T1]<-T2<-T3), monoid, base\n";
474 #endif
475 #ifndef NDEBUG
476         const bool should_not_call_eWiseApplyMonAMSS_base = false;
477         assert( should_not_call_eWiseApplyMonAMSS_base );
478 #endif
479         (void) z;
480         (void) mask;
481         (void) alpha;
482         (void) beta;
483         (void) monoid;
484         (void) phase;
485         return UNSUPPORTED;
486     }
487 
563     template<
564         Descriptor descr = descriptors::no_operation,
565         class OP, enum Backend backend,
566         typename OutputType, typename InputType1, typename InputType2,
567         typename Coords
568     >
569     RC eWiseApply(
570         Vector< OutputType, backend, Coords > &z,
571         const InputType1 alpha,
572         const Vector< InputType2, backend, Coords > &y,
573         const OP &op = OP(),
574         const Phase &phase = EXECUTE,
575         const typename std::enable_if<
576             !grb::is_object< OutputType >::value &&
577             !grb::is_object< InputType1 >::value &&
578             !grb::is_object< InputType2 >::value &&
579             grb::is_operator< OP >::value, void
580         >::type * const = nullptr
581     ) {
582 #ifdef _DEBUG
583         std::cout << "In eWiseApply ([T1]<-T2<-[T3]), operator, base\n";
584 #endif
585 #ifndef NDEBUG
586         const bool should_not_call_eWiseApplyOpASA_base = false;
587         assert( should_not_call_eWiseApplyOpASA_base );
588 #endif
589         (void) z;
590         (void) alpha;
591         (void) y;
592         (void) op;
593         (void) phase;
594         return UNSUPPORTED;
595     }
596 
672     template<
673         Descriptor descr = descriptors::no_operation,
674         class OP, enum Backend backend,
675         typename OutputType, typename MaskType,
676         typename InputType1, typename InputType2,
677         typename Coords
678     >
679     RC eWiseApply(
680         Vector< OutputType, backend, Coords > &z,
```

```
681         const Vector< MaskType, backend, Coords > &mask,
682         const InputType1 alpha,
683         const Vector< InputType2, backend, Coords > &y,
684         const OP &op = OP(),
685         const Phase &phase = EXECUTE,
686         const typename std::enable_if<
687             !grb::is_object< OutputType >::value &&
688             !grb::is_object< MaskType >::value &&
689             !grb::is_object< InputType1 >::value &&
690             !grb::is_object< InputType2 >::value &&
691             grb::is_operator< OP >::value, void
692         >::type * const = nullptr
693     ) {
694 #ifdef _DEBUG
695         std::cout << "In masked eWiseApply ([T1]<-T2<-[T3], operator, base)\n";
696 #endif
697 #ifndef NDEBUG
698         const bool should_not_call_eWiseApplyOpAMSA_base = false;
699         assert( should_not_call_eWiseApplyOpAMSA_base );
700 #endif
701         (void) z;
702         (void) mask;
703         (void) alpha;
704         (void) y;
705         (void) op;
706         (void) phase;
707         return UNSUPPORTED;
708     }
709
784     template<
785         Descriptor descr = descriptors::no_operation,
786         class Monoid, enum Backend backend,
787         typename OutputType, typename InputType1, typename InputType2,
788         typename Coords
789     >
790     RC eWiseApply(
791         Vector< OutputType, backend, Coords > &z,
792         const InputType1 alpha,
793         const Vector< InputType2, backend, Coords > &y,
794         const Monoid &monoid = Monoid(),
795         const Phase &phase = EXECUTE,
796         const typename std::enable_if< !grb::is_object< OutputType >::value &&
797             !grb::is_object< InputType1 >::value &&
798             !grb::is_object< InputType2 >::value &&
799             grb::is_monoid< Monoid >::value, void
800         >::type * const = nullptr
801     ) {
802 #ifdef _DEBUG
803         std::cout << "In unmasked eWiseApply ([T1]<-T2<-[T3], monoid, base)\n";
804 #endif
805 #ifndef NDEBUG
806         const bool should_not_call_eWiseApplyMonoidASA_base = false;
807         assert( should_not_call_eWiseApplyMonoidASA_base );
808 #endif
809         (void) z;
810         (void) alpha;
811         (void) y;
812         (void) monoid;
813         (void) phase;
814         return UNSUPPORTED;
815     }
816
895     template<
896         Descriptor descr = descriptors::no_operation,
897         class Monoid, enum Backend backend,
898         typename OutputType, typename MaskType,
899         typename InputType1, typename InputType2,
900         typename Coords
901     >
902     RC eWiseApply(
903         Vector< OutputType, backend, Coords > &z,
904         const Vector< MaskType, backend, Coords > &mask,
905         const InputType1 alpha,
906         const Vector< InputType2, backend, Coords > &y,
907         const Monoid &monoid = Monoid(),
908         const Phase &phase = EXECUTE,
909         const typename std::enable_if< !grb::is_object< OutputType >::value &&
910             !grb::is_object< MaskType >::value &&
911             !grb::is_object< InputType1 >::value &&
912             !grb::is_object< InputType2 >::value &&
913             grb::is_monoid< Monoid >::value,
914         void >::type * const = nullptr
915     ) {
916 #ifdef _DEBUG
917         std::cout << "In masked eWiseApply ([T1]<-T2<-[T3], using monoid)\n";
918 #endif
919 #ifndef NDEBUG
```

```
920         const bool should_not_call_eWiseApplyMonoidAMSA_base = false;
921         assert( should_not_call_eWiseApplyMonoidAMSA_base );
922 #endif
923         (void) z;
924         (void) mask;
925         (void) alpha;
926         (void) y;
927         (void) monoid;
928         (void) phase;
929         return UNSUPPORTED;
930     }
931
1008    template<
1009        Descriptor descr = descriptors::no_operation,
1010        class OP, enum Backend backend,
1011        typename OutputType, typename InputType1, typename InputType2,
1012        typename Coords
1013    >
1014    RC eWiseApply(
1015        Vector< OutputType, backend, Coords > &z,
1016        const Vector< InputType1, backend, Coords > &x,
1017        const InputType2 beta,
1018        const OP &op = OP(),
1019        const Phase &phase = EXECUTE,
1020        const typename std::enable_if<
1021            !grb::is_object< OutputType >::value &&
1022            !grb::is_object< InputType1 >::value &&
1023            !grb::is_object< InputType2 >::value &&
1024            grb::is_operator< OP >::value, void
1025        >::type * const = nullptr
1026    ) {
1027 #ifdef _DEBUG
1028        std::cout « "In eWiseApply ([T1]<-[T2]<-T3), operator, base\n";
1029 #endif
1030 #ifndef NDEBUG
1031        const bool should_not_call_eWiseApplyOpAAS_base = false;
1032        assert( should_not_call_eWiseApplyOpAAS_base );
1033 #endif
1034        (void) z;
1035        (void) x;
1036        (void) beta;
1037        (void) op;
1038        (void) phase;
1039        return UNSUPPORTED;
1040    }
1041
1119    template<
1120        Descriptor descr = descriptors::no_operation,
1121        class OP, enum Backend backend,
1122        typename OutputType, typename MaskType,
1123        typename InputType1, typename InputType2,
1124        typename Coords
1125    >
1126    RC eWiseApply(
1127        Vector< OutputType, backend, Coords > &z,
1128        const Vector< MaskType, backend, Coords > &mask,
1129        const Vector< InputType1, backend, Coords > &x,
1130        const InputType2 beta,
1131        const OP &op = OP(),
1132        const Phase &phase = EXECUTE,
1133        const typename std::enable_if< !grb::is_object< OutputType >::value &&
1134            !grb::is_object< MaskType >::value &&
1135            !grb::is_object< InputType1 >::value &&
1136            !grb::is_object< InputType2 >::value &&
1137            grb::is_operator< OP >::value, void
1138        >::type * const = nullptr
1139    ) {
1140 #ifdef _DEBUG
1141        std::cout « "In masked eWiseApply ([T1]<-[T2]<-T3, operator, base)\n";
1142 #endif
1143 #ifndef NDEBUG
1144        const bool should_not_call_eWiseApplyOpAMAS_base = false;
1145        assert( should_not_call_eWiseApplyOpAMAS_base );
1146 #endif
1147        (void) z;
1148        (void) mask;
1149        (void) x;
1150        (void) beta;
1151        (void) op;
1152        (void) phase;
1153        return UNSUPPORTED;
1154    }
1155
1230    template<
1231        Descriptor descr = descriptors::no_operation,
1232        class Monoid, enum Backend backend,
1233        typename OutputType, typename InputType1, typename InputType2,
```

```
1234         typename Coords
1235     >
1236     RC eWiseApply(
1237         Vector< OutputType, backend, Coords > &z,
1238         const Vector< InputType1, backend, Coords > &x,
1239         const InputType2 beta,
1240         const Monoid &monoid = Monoid(),
1241         const Phase &phase = EXECUTE,
1242         const typename std::enable_if< !grb::is_object< OutputType >::value &&
1243                 !grb::is_object< InputType1 >::value &&
1244                 !grb::is_object< InputType2 >::value &&
1245                 grb::is_monoid< Monoid >::value,
1246             void >::type * const = nullptr
1247     ) {
1248 #ifdef _DEBUG
1249         std::cout « "In unmasked eWiseApply ([T1]<-[T2]<-T3, monoid, base)\n";
1250 #endif
1251 #ifndef NDEBUG
1252         const bool should_not_call_eWiseApplyMonoidAAS_base = false;
1253         assert( should_not_call_eWiseApplyMonoidAAS_base );
1254 #endif
1255         (void) z;
1256         (void) x;
1257         (void) beta;
1258         (void) monoid;
1259         (void) phase;
1260         return UNSUPPORTED;
1261     }
1262 
1341     template<
1342         Descriptor descr = descriptors::no_operation,
1343         class Monoid, enum Backend backend,
1344         typename OutputType, typename MaskType,
1345         typename InputType1, typename InputType2,
1346         typename Coords
1347     >
1348     RC eWiseApply(
1349         Vector< OutputType, backend, Coords > &z,
1350         const Vector< MaskType, backend, Coords > &mask,
1351         const Vector< InputType1, backend, Coords > &x,
1352         const InputType2 beta,
1353         const Monoid &monoid = Monoid(),
1354         const Phase &phase = EXECUTE,
1355         const typename std::enable_if< !grb::is_object< OutputType >::value &&
1356             !grb::is_object< MaskType >::value &&
1357             !grb::is_object< InputType1 >::value &&
1358             !grb::is_object< InputType2 >::value &&
1359             grb::is_monoid< Monoid >::value, void
1360         >::type * const = nullptr
1361     ) {
1362 #ifdef _DEBUG
1363         std::cout « "In masked eWiseApply ([T1]<-[T2]<-T3, monoid, base)\n";
1364 #endif
1365 #ifndef NDEBUG
1366         const bool should_not_call_eWiseApplyMonoidAMAS_base = false;
1367         assert( should_not_call_eWiseApplyMonoidAMAS_base );
1368 #endif
1369         (void) z;
1370         (void) mask;
1371         (void) x;
1372         (void) beta;
1373         (void) monoid;
1374         (void) phase;
1375         return UNSUPPORTED;
1376     }
1377 
1454     template<
1455         Descriptor descr = descriptors::no_operation,
1456         class OP, enum Backend backend,
1457         typename OutputType, typename InputType1, typename InputType2,
1458         typename Coords
1459     >
1460     RC eWiseApply(
1461         Vector< OutputType, backend, Coords > &z,
1462         const Vector< InputType1, backend, Coords > &x,
1463         const Vector< InputType2, backend, Coords > &y,
1464         const OP &op = OP(),
1465         const Phase &phase = EXECUTE,
1466         const typename std::enable_if< !grb::is_object< OutputType >::value &&
1467             !grb::is_object< InputType1 >::value &&
1468             !grb::is_object< InputType2 >::value &&
1469             grb::is_operator< OP >::value, void
1470         >::type * const = nullptr
1471     ) {
1472 #ifdef _DEBUG
1473         std::cout « "In eWiseApply ([T1]<-[T2]<-[T3]), operator variant\n";
1474 #endif
```

```
1475 #ifndef NDEBUG
1476         const bool should_not_call_eWiseApplyOpAAA_base = false;
1477         assert( should_not_call_eWiseApplyOpAAA_base );
1478 #endif
1479         (void) z;
1480         (void) x;
1481         (void) y;
1482         (void) op;
1483         (void) phase;
1484         return UNSUPPORTED;
1485     }
1486
1566     template<
1567         Descriptor descr = descriptors::no_operation,
1568         class OP, enum Backend backend,
1569         typename OutputType, typename MaskType,
1570         typename InputType1, typename InputType2,
1571         typename Coords
1572     >
1573     RC eWiseApply(
1574         Vector< OutputType, backend, Coords > &z,
1575         const Vector< MaskType, backend, Coords > &mask,
1576         const Vector< InputType1, backend, Coords > &x,
1577         const Vector< InputType2, backend, Coords > &y,
1578         const OP &op = OP(),
1579         const Phase &phase = EXECUTE,
1580         const typename std::enable_if<
1581             !grb::is_object< OutputType >::value &&
1582             !grb::is_object< MaskType >::value &&
1583             !grb::is_object< InputType1 >::value &&
1584             !grb::is_object< InputType2 >::value &&
1585             grb::is_operator< OP >::value, void
1586         >::type * const = nullptr
1587     ) {
1588 #ifdef _DEBUG
1589         std::cout « "In masked eWiseApply ([T1]<-[T2]<-[T3], operator, base)\n";
1590 #endif
1591 #ifndef NDEBUG
1592         const bool should_not_call_eWiseApplyOpAMAA_base = false;
1593         assert( should_not_call_eWiseApplyOpAMAA_base );
1594 #endif
1595         (void) z;
1596         (void) mask;
1597         (void) x;
1598         (void) y;
1599         (void) op;
1600         (void) phase;
1601         return UNSUPPORTED;
1602     }
1603
1681     template<
1682         Descriptor descr = descriptors::no_operation,
1683         class Monoid, enum Backend backend,
1684         typename OutputType, typename InputType1, typename InputType2,
1685         typename Coords
1686     >
1687     RC eWiseApply(
1688         Vector< OutputType, backend, Coords > &z,
1689         const Vector< InputType1, backend, Coords > &x,
1690         const Vector< InputType2, backend, Coords > &y,
1691         const Monoid &monoid = Monoid(),
1692         const Phase &phase = EXECUTE,
1693         const typename std::enable_if< !grb::is_object< OutputType >::value &&
1694             !grb::is_object< InputType1 >::value &&
1695             !grb::is_object< InputType2 >::value &&
1696             grb::is_monoid< Monoid >::value, void
1697         >::type * const = nullptr
1698     ) {
1699 #ifdef _DEBUG
1700         std::cout « "In unmasked eWiseApply ([T1]<-[T2]<-[T3], monoid, base)\n";
1701 #endif
1702 #ifndef NDEBUG
1703         const bool should_not_call_eWiseApplyOpAMAA_base = false;
1704         assert( should_not_call_eWiseApplyOpAMAA_base );
1705 #endif
1706         (void) z;
1707         (void) x;
1708         (void) y;
1709         (void) monoid;
1710         (void) phase;
1711         return UNSUPPORTED;
1712     }
1713
1789     template<
1790         Descriptor descr = descriptors::no_operation,
1791         class Monoid, enum Backend backend,
1792         typename OutputType, typename MaskType,
```

```
1793         typename InputType1, typename InputType2,
1794         typename Coords
1795     >
1796     RC eWiseApply(
1797         Vector< OutputType, backend, Coords > &z,
1798         const Vector< MaskType, backend, Coords > &mask,
1799         const Vector< InputType1, backend, Coords > &x,
1800         const Vector< InputType2, backend, Coords > &y,
1801         const Monoid &monoid = Monoid(),
1802         const Phase &phase = EXECUTE,
1803         const typename std::enable_if<
1804             !grb::is_object< OutputType >::value &&
1805             !grb::is_object< MaskType >::value &&
1806             !grb::is_object< InputType1 >::value &&
1807             !grb::is_object< InputType2 >::value &&
1808             grb::is_monoid< Monoid >::value, void
1809         >::type * const = nullptr
1810     ) {
1811 #ifdef _DEBUG
1812         std::cout « "In masked eWiseApply ([T1]<-[T2]<-[T3], monoid, base)\n";
1813 #endif
1814 #ifndef NDEBUG
1815         const bool should_not_call_eWiseApplyMonoidAMAA_base = false;
1816         assert( should_not_call_eWiseApplyMonoidAMAA_base );
1817 #endif
1818         (void) z;
1819         (void) mask;
1820         (void) x;
1821         (void) y;
1822         (void) monoid;
1823         (void) phase;
1824         return UNSUPPORTED;
1825     }
1826
1906     template<
1907         Descriptor descr = descriptors::no_operation,
1908         class Ring, enum Backend backend,
1909         typename OutputType, typename InputType1, typename InputType2,
1910         typename Coords
1911     >
1912     RC eWiseAdd(
1913         Vector< OutputType, backend, Coords > &z,
1914         const Vector< InputType1, backend, Coords > &x,
1915         const Vector< InputType2, backend, Coords > &y,
1916         const Ring &ring = Ring(),
1917         const Phase &phase = EXECUTE,
1918         const typename std::enable_if< !grb::is_object< OutputType >::value &&
1919             !grb::is_object< InputType1 >::value &&
1920             !grb::is_object< InputType2 >::value &&
1921             grb::is_semiring< Ring >::value, void
1922         >::type * const = nullptr
1923     ) {
1924 #ifdef _DEBUG
1925         std::cout « "in eWiseAdd ([T1] <- [T2] + [T3]), unmasked, base";
1926 #endif
1927 #ifndef NDEBUG
1928         const bool should_not_call_eWiseAddAAA_base = false;
1929         assert( should_not_call_eWiseAddAAA_base );
1930 #endif
1931         (void) z;
1932         (void) x;
1933         (void) y;
1934         (void) ring;
1935         (void) phase;
1936         return UNSUPPORTED;
1937     }
1938
2013     template<
2014         Descriptor descr = descriptors::no_operation,
2015         class Ring, enum Backend backend,
2016         typename InputType1, typename InputType2, typename OutputType,
2017         typename Coords
2018     >
2019     RC eWiseAdd(
2020         Vector< OutputType, backend, Coords > &z,
2021         const InputType1 alpha,
2022         const Vector< InputType2, backend, Coords > &y,
2023         const Ring &ring = Ring(),
2024         const Phase &phase = EXECUTE,
2025         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2026             !grb::is_object< InputType1 >::value &&
2027             !grb::is_object< InputType2 >::value &&
2028             grb::is_semiring< Ring >::value, void
2029         >::type * const = nullptr
2030     ) {
2031 #ifdef _DEBUG
2032         std::cout « "in eWiseAdd ([T1] <- T2 + [T3]), unmasked, base";
```

```
2033 #endif
2034 #ifndef NDEBUG
2035         const bool should_not_call_eWiseAddASA_base = false;
2036         assert( should_not_call_eWiseAddASA_base );
2037 #endif
2038         (void) z;
2039         (void) alpha;
2040         (void) y;
2041         (void) ring;
2042         (void) phase;
2043         return UNSUPPORTED;
2044     }
2045
2120     template<
2121         Descriptor descr = descriptors::no_operation,
2122         class Ring, enum Backend backend,
2123         typename InputType1, typename InputType2, typename OutputType,
2124         typename Coords
2125     >
2126     RC eWiseAdd(
2127         Vector< OutputType, backend, Coords > &z,
2128         const Vector< InputType1, backend, Coords > &x,
2129         const InputType2 beta,
2130         const Ring &ring = Ring(),
2131         const Phase &phase = EXECUTE,
2132         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2133             !grb::is_object< InputType1 >::value &&
2134             !grb::is_object< InputType2 >::value &&
2135             grb::is_semiring< Ring >::value, void
2136         >::type * const = nullptr
2137     ) {
2138 #ifdef _DEBUG
2139         std::cout « "in eWiseAdd ([T1] <- [T2] + T3), unmasked, base";
2140 #endif
2141 #ifndef NDEBUG
2142         const bool should_not_call_eWiseAddAAS_base = false;
2143         assert( should_not_call_eWiseAddAAS_base );
2144 #endif
2145         (void) z;
2146         (void) x;
2147         (void) beta;
2148         (void) ring;
2149         (void) phase;
2150         return UNSUPPORTED;
2151     }
2152
2223     template<
2224         Descriptor descr = descriptors::no_operation,
2225         class Ring, enum Backend backend,
2226         typename InputType1, typename InputType2, typename OutputType,
2227         typename Coords
2228     >
2229     RC eWiseAdd(
2230         Vector< OutputType, backend, Coords > &z,
2231         const InputType1 alpha,
2232         const InputType2 beta,
2233         const Ring &ring = Ring(),
2234         const Phase &phase = EXECUTE,
2235         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2236             !grb::is_object< InputType1 >::value &&
2237             !grb::is_object< InputType2 >::value &&
2238             grb::is_semiring< Ring >::value, void
2239         >::type * const = nullptr
2240     ) {
2241 #ifdef _DEBUG
2242         std::cout « "in eWiseAdd ([T1] <- T2 + T3), unmasked, base";
2243 #endif
2244 #ifndef NDEBUG
2245         const bool should_not_call_eWiseAddASS_base = false;
2246         assert( should_not_call_eWiseAddASS_base );
2247 #endif
2248         (void) z;
2249         (void) alpha;
2250         (void) beta;
2251         (void) ring;
2252         (void) phase;
2253         return UNSUPPORTED;
2254     }
2255
2341     template<
2342         Descriptor descr = descriptors::no_operation,
2343         class Ring, enum Backend backend,
2344         typename OutputType, typename MaskType,
2345         typename InputType1, typename InputType2,
2346         typename Coords
2347     >
2348     RC eWiseAdd(
```

```
2349         Vector< OutputType, backend, Coords > &z,
2350         const Vector< MaskType, backend, Coords > &mask,
2351         const Vector< InputType1, backend, Coords > &x,
2352         const Vector< InputType2, backend, Coords > &y,
2353         const Ring &ring = Ring(),
2354         const Phase &phase = EXECUTE,
2355         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2356             !grb::is_object< InputType1 >::value &&
2357             !grb::is_object< InputType2 >::value &&
2358             grb::is_semiring< Ring >::value, void
2359         >::type * const = nullptr
2360     ) {
2361 #ifdef _DEBUG
2362         std::cout « "in eWiseAdd ([T1] <- [T2] + [T3]), masked, base";
2363 #endif
2364 #ifndef NDEBUG
2365         const bool should_not_call_eWiseAddAMAA_base = false;
2366         assert( should_not_call_eWiseAddAMAA_base );
2367 #endif
2368         (void) z;
2369         (void) mask;
2370         (void) x;
2371         (void) y;
2372         (void) ring;
2373         (void) phase;
2374         return UNSUPPORTED;
2375     }
2376
2458     template<
2459         Descriptor descr = descriptors::no_operation,
2460         class Ring, enum Backend backend,
2461         typename InputType1, typename InputType2,
2462         typename OutputType, typename MaskType,
2463         typename Coords
2464     >
2465     RC eWiseAdd(
2466         Vector< OutputType, backend, Coords > &z,
2467         const Vector< MaskType, backend, Coords > &mask,
2468         const InputType1 alpha,
2469         const Vector< InputType2, backend, Coords > &y,
2470         const Ring &ring = Ring(),
2471         const Phase &phase = EXECUTE,
2472         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2473             !grb::is_object< InputType1 >::value &&
2474             !grb::is_object< InputType2 >::value &&
2475             grb::is_semiring< Ring >::value, void
2476         >::type * const = nullptr
2477     ) {
2478 #ifdef _DEBUG
2479         std::cout « "in eWiseAdd ([T1] <- T2 + [T3]), masked, base";
2480 #endif
2481 #ifndef NDEBUG
2482         const bool should_not_call_eWiseAddAMSA_base = false;
2483         assert( should_not_call_eWiseAddAMSA_base );
2484 #endif
2485         (void) z;
2486         (void) mask;
2487         (void) alpha;
2488         (void) y;
2489         (void) ring;
2490         (void) phase;
2491         return UNSUPPORTED;
2492     }
2493
2575     template<
2576         Descriptor descr = descriptors::no_operation,
2577         class Ring, enum Backend backend,
2578         typename InputType1, typename InputType2,
2579         typename OutputType, typename MaskType,
2580         typename Coords
2581     >
2582     RC eWiseAdd(
2583         Vector< OutputType, backend, Coords > &z,
2584         const Vector< MaskType, backend, Coords > &mask,
2585         const Vector< InputType1, backend, Coords > &x,
2586         const InputType2 beta,
2587         const Ring &ring = Ring(),
2588         const Phase &phase = EXECUTE,
2589         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2590             !grb::is_object< InputType1 >::value &&
2591             !grb::is_object< InputType2 >::value &&
2592             grb::is_semiring< Ring >::value, void
2593         >::type * const = nullptr
2594     ) {
2595 #ifdef _DEBUG
2596         std::cout « "in eWiseAdd ([T1] <- [T2] + T3), masked, base";
2597 #endif
```

```
2598 #ifndef NDEBUG
2599         const bool should_not_call_eWiseAddAMAS_base = false;
2600         assert( should_not_call_eWiseAddAMAS_base );
2601 #endif
2602         (void) z;
2603         (void) mask;
2604         (void) x;
2605         (void) beta;
2606         (void) ring;
2607         (void) phase;
2608         return UNSUPPORTED;
2609     }
2610 
2688     template<
2689         Descriptor descr = descriptors::no_operation,
2690         class Ring, enum Backend backend,
2691         typename InputType1, typename InputType2,
2692         typename OutputType, typename MaskType,
2693         typename Coords
2694     >
2695     RC eWiseAdd(
2696         Vector< OutputType, backend, Coords > &z,
2697         const Vector< MaskType, backend, Coords > &mask,
2698         const InputType1 alpha,
2699         const InputType2 beta,
2700         const Ring &ring = Ring(),
2701         const Phase &phase = EXECUTE,
2702         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2703             !grb::is_object< InputType1 >::value &&
2704             !grb::is_object< InputType2 >::value &&
2705             grb::is_semiring< Ring >::value, void
2706         >::type * const = nullptr
2707     ) {
2708 #ifdef _DEBUG
2709         std::cout « "in eWiseAdd ([T1] <- T2 + T3), masked, base";
2710 #endif
2711 #ifndef NDEBUG
2712         const bool should_not_call_eWiseAddAMSS_base = false;
2713         assert( should_not_call_eWiseAddAMSS_base );
2714 #endif
2715         (void) z;
2716         (void) mask;
2717         (void) alpha;
2718         (void) beta;
2719         (void) ring;
2720         (void) phase;
2721         return UNSUPPORTED;
2722     }
2723 
2788     template<
2789         Descriptor descr = descriptors::no_operation,
2790         class Ring, enum Backend backend,
2791         typename InputType1, typename InputType2, typename OutputType,
2792         typename Coords
2793     >
2794     RC eWiseMul(
2795         Vector< OutputType, backend, Coords > &z,
2796         const Vector< InputType1, backend, Coords > &x,
2797         const Vector< InputType2, backend, Coords > &y,
2798         const Ring &ring = Ring(),
2799         const Phase &phase = EXECUTE,
2800         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2801             !grb::is_object< InputType1 >::value &&
2802             !grb::is_object< InputType2 >::value &&
2803             grb::is_semiring< Ring >::value, void
2804         >::type * const = nullptr
2805     ) {
2806 #ifdef _DEBUG
2807         std::cout « "in eWiseMul ([T1] <- [T2] * [T3]), unmasked, base";
2808 #endif
2809 #ifndef NDEBUG
2810         const bool should_not_call_eWiseMulAAA_base = false;
2811         assert( should_not_call_eWiseMulAAA_base );
2812 #endif
2813         (void) z;
2814         (void) x;
2815         (void) y;
2816         (void) ring;
2817         (void) phase;
2818         return UNSUPPORTED;
2819     }
2820 
2885     template<
2886         Descriptor descr = descriptors::no_operation,
2887         class Ring, enum Backend backend,
2888         typename InputType1, typename InputType2, typename OutputType,
2889         typename Coords
```

```
2890     >
2891     RC eWiseMul(
2892         Vector< OutputType, backend, Coords > &z,
2893         const InputType1 alpha,
2894         const Vector< InputType2, backend, Coords > &y,
2895         const Ring &ring = Ring(),
2896         const Phase &phase = EXECUTE,
2897         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2898             !grb::is_object< InputType1 >::value &&
2899             !grb::is_object< InputType2 >::value &&
2900             grb::is_semiring< Ring >::value, void
2901         >::type * const = nullptr
2902     ) {
2903 #ifdef _DEBUG
2904         std::cout « "in eWiseMul ([T1] <- T2 * [T3]), unmasked, base";
2905 #endif
2906 #ifndef NDEBUG
2907         const bool should_not_call_eWiseMulASA_base = false;
2908         assert( should_not_call_eWiseMulASA_base );
2909 #endif
2910         (void) z;
2911         (void) alpha;
2912         (void) y;
2913         (void) ring;
2914         (void) phase;
2915         return UNSUPPORTED;
2916     }
2917
2982     template<
2983         Descriptor descr = descriptors::no_operation,
2984         class Ring, enum Backend backend,
2985         typename InputType1, typename InputType2, typename OutputType,
2986         typename Coords
2987     >
2988     RC eWiseMul(
2989         Vector< OutputType, backend, Coords > &z,
2990         const Vector< InputType1, backend, Coords > &x,
2991         const InputType2 beta,
2992         const Ring &ring = Ring(),
2993         const Phase &phase = EXECUTE,
2994         const typename std::enable_if< !grb::is_object< OutputType >::value &&
2995             !grb::is_object< InputType1 >::value &&
2996             !grb::is_object< InputType2 >::value &&
2997             grb::is_semiring< Ring >::value, void
2998         >::type * const = nullptr
2999     ) {
3000 #ifdef _DEBUG
3001         std::cout « "in eWiseMul ([T1] <- [T2] * T3), unmasked, base";
3002 #endif
3003 #ifndef NDEBUG
3004         const bool should_not_call_eWiseMulAAS_base = false;
3005         assert( should_not_call_eWiseMulAAS_base );
3006 #endif
3007         (void) z;
3008         (void) x;
3009         (void) beta;
3010         (void) ring;
3011         (void) phase;
3012         return UNSUPPORTED;
3013     }
3014
3075     template<
3076         Descriptor descr = descriptors::no_operation,
3077         class Ring, enum Backend backend,
3078         typename InputType1, typename InputType2, typename OutputType,
3079         typename Coords
3080     >
3081     RC eWiseMul(
3082         Vector< OutputType, backend, Coords > &z,
3083         const InputType1 alpha,
3084         const InputType2 beta,
3085         const Ring &ring = Ring(),
3086         const Phase &phase = EXECUTE,
3087         const typename std::enable_if< !grb::is_object< OutputType >::value &&
3088             !grb::is_object< InputType1 >::value &&
3089             !grb::is_object< InputType2 >::value &&
3090             grb::is_semiring< Ring >::value, void
3091         >::type * const = nullptr
3092     ) {
3093 #ifdef _DEBUG
3094         std::cout « "in eWiseMul ([T1] <- T2 * T3), unmasked, base";
3095 #endif
3096 #ifndef NDEBUG
3097         const bool should_not_call_eWiseMulASS_base = false;
3098         assert( should_not_call_eWiseMulASS_base );
3099 #endif
3100         (void) z;
```

```
3101         (void) alpha;
3102         (void) beta;
3103         (void) ring;
3104         (void) phase;
3105         return UNSUPPORTED;
3106     }
3107 
3177     template<
3178         Descriptor descr = descriptors::no_operation,
3179         class Ring, enum Backend backend,
3180         typename InputType1, typename InputType2,
3181         typename OutputType, typename MaskType,
3182         typename Coords
3183     >
3184     RC eWiseMul(
3185         Vector< OutputType, backend, Coords > &z,
3186         const Vector< MaskType, backend, Coords > &mask,
3187         const Vector< InputType1, backend, Coords > &x,
3188         const Vector< InputType2, backend, Coords > &y,
3189         const Ring &ring = Ring(),
3190         const Phase &phase = EXECUTE,
3191         const typename std::enable_if< !grb::is_object< OutputType >::value &&
3192             !grb::is_object< InputType1 >::value &&
3193             !grb::is_object< InputType2 >::value &&
3194             grb::is_semiring< Ring >::value, void
3195         >::type * const = nullptr
3196     ) {
3197 #ifdef _DEBUG
3198         std::cout « "in eWiseMul ([T1] <- [T2] * [T3]), masked, base";
3199 #endif
3200 #ifndef NDEBUG
3201         const bool should_not_call_eWiseMulAMAA_base = false;
3202         assert( should_not_call_eWiseMulAMAA_base );
3203 #endif
3204         (void) z;
3205         (void) mask;
3206         (void) x;
3207         (void) y;
3208         (void) ring;
3209         (void) phase;
3210         return UNSUPPORTED;
3211     }
3212 
3282     template<
3283         Descriptor descr = descriptors::no_operation,
3284         class Ring, enum Backend backend,
3285         typename InputType1, typename InputType2,
3286         typename OutputType, typename MaskType,
3287         typename Coords
3288     >
3289     RC eWiseMul(
3290         Vector< OutputType, backend, Coords > &z,
3291         const Vector< MaskType, backend, Coords > &mask,
3292         const InputType1 alpha,
3293         const Vector< InputType2, backend, Coords > &y,
3294         const Ring &ring = Ring(),
3295         const Phase &phase = EXECUTE,
3296         const typename std::enable_if< !grb::is_object< OutputType >::value &&
3297             !grb::is_object< InputType1 >::value &&
3298             !grb::is_object< InputType2 >::value &&
3299             grb::is_semiring< Ring >::value, void
3300         >::type * const = nullptr
3301     ) {
3302 #ifdef _DEBUG
3303         std::cout « "in eWiseMul ([T1] <- T2 * [T3]), masked, base";
3304 #endif
3305 #ifndef NDEBUG
3306         const bool should_not_call_eWiseMulAMSA_base = false;
3307         assert( should_not_call_eWiseMulAMSA_base );
3308 #endif
3309         (void) z;
3310         (void) mask;
3311         (void) alpha;
3312         (void) y;
3313         (void) ring;
3314         (void) phase;
3315         return UNSUPPORTED;
3316     }
3317 
3387     template<
3388         Descriptor descr = descriptors::no_operation,
3389         class Ring, enum Backend backend,
3390         typename InputType1, typename InputType2,
3391         typename OutputType, typename MaskType,
3392         typename Coords
3393     >
3394     RC eWiseMul(
```

```
3395         Vector< OutputType, backend, Coords > &z,
3396         const Vector< MaskType, backend, Coords > &mask,
3397         const Vector< InputType1, backend, Coords > &x,
3398         const InputType2 beta,
3399         const Ring &ring = Ring(),
3400         const Phase &phase = EXECUTE,
3401         const typename std::enable_if< !grb::is_object< OutputType >::value &&
3402             !grb::is_object< InputType1 >::value &&
3403             !grb::is_object< InputType2 >::value &&
3404             grb::is_semiring< Ring >::value, void
3405         >::type * const = nullptr
3406     ) {
3407 #ifdef _DEBUG
3408         std::cout « "in eWiseMul ([T1] <- [T2] * T3), masked, base";
3409 #endif
3410 #ifndef NDEBUG
3411         const bool should_not_call_eWiseMulAMAS_base = false;
3412         assert( should_not_call_eWiseMulAMAS_base );
3413 #endif
3414         (void) z;
3415         (void) mask;
3416         (void) x;
3417         (void) beta;
3418         (void) ring;
3419         (void) phase;
3420         return UNSUPPORTED;
3421     }
3422
3489     template<
3490         Descriptor descr = descriptors::no_operation,
3491         class Ring, enum Backend backend,
3492         typename InputType1, typename InputType2,
3493         typename OutputType, typename MaskType,
3494         typename Coords
3495     >
3496     RC eWiseMul(
3497         Vector< OutputType, backend, Coords > &z,
3498         const Vector< MaskType, backend, Coords > &mask,
3499         const InputType1 alpha,
3500         const InputType2 beta,
3501         const Ring &ring = Ring(),
3502         const Phase &phase = EXECUTE,
3503         const typename std::enable_if< !grb::is_object< OutputType >::value &&
3504             !grb::is_object< InputType1 >::value &&
3505             !grb::is_object< InputType2 >::value &&
3506             grb::is_semiring< Ring >::value, void
3507         >::type * const = nullptr
3508     ) {
3509 #ifdef _DEBUG
3510         std::cout « "in eWiseMul ([T1] <- T2 * T3), masked, base";
3511 #endif
3512 #ifndef NDEBUG
3513         const bool should_not_call_eWiseMulAMSS_base = false;
3514         assert( should_not_call_eWiseMulAMSS_base );
3515 #endif
3516         (void) z;
3517         (void) mask;
3518         (void) alpha;
3519         (void) beta;
3520         (void) ring;
3521         (void) phase;
3522         return UNSUPPORTED;
3523     }
3524
3739     template<
3740         typename Func,
3741         typename DataType,
3742         Backend backend,
3743         typename Coords,
3744         typename... Args
3745     >
3746     RC eWiseLambda(
3747         const Func f,
3748         const Vector< DataType, backend, Coords > & x, Args...
3749     ) {
3750 #ifndef NDEBUG
3751         const bool should_not_call_base_vector_ewiselambda = false;
3752         assert( should_not_call_base_vector_ewiselambda );
3753 #endif
3754         (void) f;
3755         (void) x;
3756         return UNSUPPORTED;
3757     }
3758
3834     template<
3835         Descriptor descr = descriptors::no_operation,
3836         class Monoid,
```

```
3837         typename InputType, typename IOType, typename MaskType,
3838         Backend backend, typename Coords
3839     >
3840     RC foldl(
3841         IOType &x,
3842         const Vector< InputType, backend, Coords > &y,
3843         const Vector< MaskType, backend, Coords > &mask,
3844         const Monoid &monoid = Monoid(),
3845         const typename std::enable_if< !grb::is_object< IOType >::value &&
3846             !grb::is_object< InputType >::value &&
3847             !grb::is_object< MaskType >::value &&
3848             grb::is_monoid< Monoid >::value, void
3849         >::type * const = nullptr
3850     ) {
3851 #ifndef NDEBUG
3852         const bool should_not_call_base_scalar_foldl = false;
3853         assert( should_not_call_base_scalar_foldl );
3854 #endif
3855         (void) y;
3856         (void) x;
3857         (void) mask;
3858         (void) monoid;
3859         return UNSUPPORTED;
3860     }
3861
3867     template<
3868         Descriptor descr = descriptors::no_operation,
3869         class Monoid,
3870         typename IOType, typename InputType,
3871         Backend backend,
3872         typename Coords
3873     >
3874     RC foldl(
3875         IOType &x,
3876         const Vector< InputType, backend, Coords > &y,
3877         const Monoid &monoid = Monoid(),
3878         const typename std::enable_if<
3879             !grb::is_object< IOType >::value &&
3880             grb::is_monoid< Monoid >::value,
3881         void >::type * const = nullptr
3882     ) {
3883 #ifndef NDEBUG
3884         const bool should_not_call_base_scalar_foldl_nomask = false;
3885         assert( should_not_call_base_scalar_foldl_nomask );
3886 #endif
3887         (void) y;
3888         (void) x;
3889         (void) monoid;
3890         return UNSUPPORTED;
3891     }
3892
3903     template<
3904         Descriptor descr = descriptors::no_operation,
3905         class OP,
3906         typename IOType, typename InputType, typename MaskType,
3907         Backend backend, typename Coords
3908     >
3909     RC foldl(
3910         IOType &x,
3911         const Vector< InputType, backend, Coords > &y,
3912         const Vector< MaskType, backend, Coords > &mask,
3913         const OP &op = OP(),
3914         const typename std::enable_if<
3915             !grb::is_object< IOType >::value &&
3916             !grb::is_object< MaskType >::value &&
3917             grb::is_operator< OP >::value,
3918         void >::type * const = nullptr
3919     ) {
3920 #ifndef NDEBUG
3921         const bool should_not_call_base_scalar_foldl_op = false;
3922         assert( should_not_call_base_scalar_foldl_op );
3923 #endif
3924         (void) x;
3925         (void) y;
3926         (void) mask;
3927         (void) op;
3928         return UNSUPPORTED;
3929     }
3930
3937     template<
3938         Descriptor descr = descriptors::no_operation,
3939         class Monoid,
3940         typename InputType, typename IOType, typename MaskType,
3941         Backend backend, typename Coords
3942     >
3943     RC foldr(
3944         const Vector< InputType, backend, Coords > &x,
```

```
3945         const Vector< MaskType, backend, Coords > &mask,
3946         IOType &y,
3947         const Monoid &monoid = Monoid(),
3948         const typename std::enable_if< !grb::is_object< IOType >::value &&
3949             !grb::is_object< InputType >::value &&
3950             !grb::is_object< MaskType >::value &&
3951             grb::is_monoid< Monoid >::value, void
3952         >::type * const = nullptr
3953     ) {
3954 #ifndef NDEBUG
3955         const bool should_not_call_base_scalar_foldr = false;
3956         assert( should_not_call_base_scalar_foldr );
3957 #endif
3958         (void) y;
3959         (void) x;
3960         (void) mask;
3961         (void) monoid;
3962         return UNSUPPORTED;
3963     }
3964
3971     template<
3972         Descriptor descr = descriptors::no_operation,
3973         class Monoid,
3974         typename IOType, typename InputType,
3975         Backend backend, typename Coords
3976     >
3977     RC foldr(
3978         const Vector< InputType, backend, Coords > &y,
3979         IOType &x,
3980         const Monoid &monoid = Monoid(),
3981         const typename std::enable_if<
3982             !grb::is_object< IOType >::value &&
3983             grb::is_monoid< Monoid >::value,
3984         void >::type * const = nullptr
3985     ) {
3986 #ifndef NDEBUG
3987         const bool should_not_call_base_scalar_foldr_nomask = false;
3988         assert( should_not_call_base_scalar_foldr_nomask );
3989 #endif
3990         (void) y;
3991         (void) x;
3992         (void) monoid;
3993         return UNSUPPORTED;
3994     }
3995
4050     template<
4051         Descriptor descr = descriptors::no_operation,
4052         class AddMonoid, class AnyOp,
4053         typename OutputType, typename InputType1, typename InputType2,
4054         enum Backend backend, typename Coords
4055     >
4056     RC dot(
4057         OutputType &z,
4058         const Vector< InputType1, backend, Coords > &x,
4059         const Vector< InputType2, backend, Coords > &y,
4060         const AddMonoid &addMonoid = AddMonoid(),
4061         const AnyOp &anyOp = AnyOp(),
4062         const Phase &phase = EXECUTE,
4063         const typename std::enable_if< !grb::is_object< OutputType >::value &&
4064             !grb::is_object< InputType1 >::value &&
4065             !grb::is_object< InputType2 >::value &&
4066             grb::is_monoid< AddMonoid >::value &&
4067             grb::is_operator< AnyOp >::value,
4068         void >::type * const = nullptr
4069     ) {
4070 #ifdef _DEBUG
4071         std::cout « "Should not call base grb::dot (monoid-operator version)\n";
4072 #endif
4073 #ifndef NDEBUG
4074         const bool should_not_call_base_dot_monOp = false;
4075         assert( should_not_call_base_dot_monOp );
4076 #endif
4077         (void) z;
4078         (void) x;
4079         (void) y;
4080         (void) addMonoid;
4081         (void) anyOp;
4082         (void) phase;
4083         return UNSUPPORTED;
4084     }
4085
4128     template<
4129         Descriptor descr = descriptors::no_operation,
4130         class Ring,
4131         typename IOType, typename InputType1, typename InputType2,
4132         Backend backend, typename Coords
4133     >
```

```
4134     RC dot(
4135         IOType &z,
4136         const Vector< InputType1, backend, Coords > &x,
4137         const Vector< InputType2, backend, Coords > &y,
4138         const Ring &ring = Ring(),
4139         const Phase &phase = EXECUTE,
4140         const typename std::enable_if<
4141             !grb::is_object< InputType1 >::value &&
4142             !grb::is_object< InputType2 >::value &&
4143             !grb::is_object< IOType >::value &&
4144             grb::is_semiring< Ring >::value,
4145         void >::type * const = nullptr
4146     ) {
4147 #ifdef _DEBUG
4148         std::cout << "Should not call base grb::dot (semiring version)\n";
4149 #endif
4150 #ifndef NDEBUG
4151         const bool should_not_call_base_dot_semiring = false;
4152         assert( should_not_call_base_dot_semiring );
4153 #endif
4154         (void) z;
4155         (void) x;
4156         (void) y;
4157         (void) ring;
4158         (void) phase;
4159         return UNSUPPORTED;
4160     }
4161
4164 } // end namespace grb
4165
4166 #endif // end _H_GRB_BASE_BLAS1
4167
```

10.37 blas2.hpp File Reference

Defines the ALP/GraphBLAS level-2 API.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Functions

- template<typename Func , typename DataType , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_backend, typename... Args>
  RC eWiseLambda (const Func f, const Matrix< DataType, implementation, RIT, CIT, NIT > &A, Args...)

  Executes an arbitrary element-wise user-defined function f on all nonzero elements of a given matrix A.
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place doubly-masked sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Matrix< InputType2, backend, RIT, NIT, CIT > &A, const Vector< InputType1, backend, Coords > &v, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Right-handed in-place masked sparse matrix–vector multiplication, $u = u + Av$, over a given commutative additive monoid and any binary operator acting as multiplication.
- template<Descriptor descr = descriptors::no_operation, class Semiring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC mxv (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &u_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Semiring &semiring=Semiring(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_semiring< Semiring >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value, void >::type *const =nullptr)

  Right-handed in-place doubly-masked sparse matrix times vector multiplication, $u = u + Av$.
- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_backend>
  RC mxv (Vector< IOType, implementation, Coords > &u, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Vector< InputType1, implementation, Coords > &v, const Ring &ring, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Right-handed in-place sparse matrix–vector multiplication, $u = u + Av$, over a given semiring.
- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , typename Coords , enum Backend implementation = config::default_backend>
  RC mxv (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords > &mask, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Vector< InputType1, implementation, Coords > &v, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Right-handed in-place masked sparse matrix–vector multiplication, $u = u + Av$, over a given semiring.
- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType1, backend, Coords > &v, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

Left-handed in-place sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &mask, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, const typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!std::is_same< InputType2, void >::value, void >::type *const =nullptr)

  Left-handed in-place doubly-masked sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

- template<Descriptor descr = descriptors::no_operation, class Semiring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename InputType4 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend backend>
  RC vxm (Vector< IOType, backend, Coords > &u, const Vector< InputType3, backend, Coords > &u_mask, const Vector< InputType1, backend, Coords > &v, const Vector< InputType4, backend, Coords > &v_mask, const Matrix< InputType2, backend, RIT, CIT, NIT > &A, const Semiring &semiring=Semiring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Semiring >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< InputType3 >::value &&!grb::is_object< InputType4 >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Left-handed in-place doubly-masked sparse matrix times vector multiplication, $u = u + vA$.

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend implementation = config::default_backend>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Left-handed in-place sparse matrix–vector multiplication, $u = u + vA$, over a given semiring.

- template<Descriptor descr = descriptors::no_operation, class AdditiveMonoid , class MultiplicativeOperator , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , Backend implementation>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords > &mask, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const AdditiveMonoid &add=AdditiveMonoid(), const MultiplicativeOperator &mul=MultiplicativeOperator(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_monoid< AdditiveMonoid >::value &&grb::is_operator< MultiplicativeOperator >::value &&!grb::is_object< IOType >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!std::is_same< InputType2, void >::value, void >::type *=nullptr)

  Left-handed in-place masked sparse matrix–vector multiplication, $u = u + vA$, over a given commutative additive monoid and any binary operator acting as multiplication.

- template<Descriptor descr = descriptors::no_operation, class Ring , typename IOType , typename InputType1 , typename InputType2 , typename InputType3 , typename Coords , typename RIT , typename CIT , typename NIT , enum Backend implementation = config::default_backend>
  RC vxm (Vector< IOType, implementation, Coords > &u, const Vector< InputType3, implementation, Coords > &mask, const Vector< InputType1, implementation, Coords > &v, const Matrix< InputType2, implementation, RIT, CIT, NIT > &A, const Ring &ring=Ring(), const Phase &phase=EXECUTE, typename std::enable_if< grb::is_semiring< Ring >::value, void >::type *=nullptr)

  Left-handed in-place masked sparse matrix–vector multiplication, $u = u + vA$, over a given semiring.

10.37.1 Detailed Description

Defines the ALP/GraphBLAS level-2 API.

Author

A. N. Yzelman

Date

30th of March 2017

10.38 blas2.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_BLAS2_BASE
28 #define _H_GRB_BLAS2_BASE
29
30 #include <assert.h>
31
32 #include <graphblas/backends.hpp>
33 #include <graphblas/blas1.hpp>
34 #include <graphblas/descriptors.hpp>
35 #include <graphblas/rc.hpp>
36 #include <graphblas/semiring.hpp>
37
38 #include "config.hpp"
39 #include "matrix.hpp"
40 #include "vector.hpp"
41
42
43 namespace grb {
44
235     template<
236         Descriptor descr = descriptors::no_operation,
237         class Semiring,
238         typename IOType, typename InputType1, typename InputType2,
239         typename InputType3, typename InputType4,
240         typename Coords, typename RIT, typename CIT, typename NIT,
241         Backend backend
242     >
243     RC mxv(
244         Vector< IOType, backend, Coords > &u,
245         const Vector< InputType3, backend, Coords > &u_mask,
246         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
247         const Vector< InputType1, backend, Coords > &v,
248         const Vector< InputType4, backend, Coords > &v_mask,
249         const Semiring &semiring = Semiring(),
250         const Phase &phase = EXECUTE,
251         const typename std::enable_if<
252             grb::is_semiring< Semiring >::value &&
253             !grb::is_object< IOType >::value &&
254             !grb::is_object< InputType1 >::value &&
255             !grb::is_object< InputType2 >::value &&
256             !grb::is_object< InputType3 >::value &&
257             !grb::is_object< InputType4 >::value,
258         void >::type * const = nullptr
259     ) {
260 #ifdef _DEBUG
261         std::cerr « "Selected backend does not implement mxv "
262             « "(doubly-masked, semiring)\n";
263 #endif
264 #ifndef NDEBUG
265         const bool selected_backed_does_not_support_doubly_masked_mxv_sr = false;
266         assert( selected_backed_does_not_support_doubly_masked_mxv_sr );
```

```
267 #endif
268         (void) u;
269         (void) u_mask;
270         (void) A;
271         (void) v;
272         (void) v_mask;
273         (void) semiring;
274         return UNSUPPORTED;
275     }
276 
299     template<
300         Descriptor descr = descriptors::no_operation,
301         class Semiring,
302         typename IOType, typename InputType1, typename InputType2,
303         typename InputType3, typename InputType4,
304         typename Coords, typename RIT, typename CIT, typename NIT,
305         enum Backend backend
306     >
307     RC vxm(
308         Vector< IOType, backend, Coords > &u,
309         const Vector< InputType3, backend, Coords > &u_mask,
310         const Vector< InputType1, backend, Coords > &v,
311         const Vector< InputType4, backend, Coords > &v_mask,
312         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
313         const Semiring &semiring = Semiring(),
314         const Phase &phase = EXECUTE,
315         typename std::enable_if<
316             grb::is_semiring< Semiring >::value &&
317             !grb::is_object< InputType1 >::value &&
318             !grb::is_object< InputType2 >::value &&
319             !grb::is_object< InputType3 >::value &&
320             !grb::is_object< InputType4 >::value &&
321             !grb::is_object< IOType >::value,
322         void >::type * = nullptr
323     ) {
324 #ifdef _DEBUG
325         std::cerr << "Selected backend does not implement doubly-masked grb::vxm\n";
326 #endif
327 #ifndef NDEBUG
328         const bool selected_backend_does_not_support_doubly_masked_vxm_sr = false;
329         assert( selected_backend_does_not_support_doubly_masked_vxm_sr );
330 #endif
331         (void) u;
332         (void) u_mask;
333         (void) v;
334         (void) v_mask;
335         (void) A;
336         (void) semiring;
337         return UNSUPPORTED;
338     }
339 
434     template<
435         typename Func, typename DataType,
436         typename RIT, typename CIT, typename NIT,
437         Backend implementation = config::default_backend,
438         typename... Args
439     >
440     RC eWiseLambda(
441         const Func f,
442         const Matrix< DataType, implementation, RIT, CIT, NIT > &A,
443         Args...
444     ) {
445 #ifdef _DEBUG
446         std::cerr << "Selected backend does not implement grb::eWiseLambda (matrices)\n";
447 #endif
448 #ifndef NDEBUG
449         const bool selected_backend_does_not_support_matrix_eWiseLamba = false;
450         assert( selected_backend_does_not_support_matrix_eWiseLamba );
451 #endif
452         (void) f;
453         (void) A;
454         return UNSUPPORTED;
455     }
456 
457      // default (non-)implementations follow:
458 
471     template<
472         Descriptor descr = descriptors::no_operation,
473         class Ring,
474         typename IOType, typename InputType1, typename InputType2,
475         typename InputType3,
476         typename RIT, typename CIT, typename NIT,
477         typename Coords,
478         enum Backend implementation = config::default_backend
479     >
480     RC mxv(
481         Vector< IOType, implementation, Coords > &u,
```

```
482         const Vector< InputType3, implementation, Coords > &mask,
483         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
484         const Vector< InputType1, implementation, Coords > &v,
485         const Ring &ring = Ring(),
486         const Phase &phase = EXECUTE,
487         typename std::enable_if<
488             grb::is_semiring< Ring >::value,
489         void >::type * = nullptr
490     ) {
491 #ifdef _DEBUG
492         std::cerr « "Selected backend does not implement grb::mxv (output-masked)\n";
493 #endif
494 #ifndef NDEBUG
495         const bool backend_does_not_support_output_masked_mxv = false;
496         assert( backend_does_not_support_output_masked_mxv );
497 #endif
498         (void) u;
499         (void) mask;
500         (void) A;
501         (void) v;
502         (void) ring;
503         return UNSUPPORTED;
504     }
505
518     template<
519         Descriptor descr = descriptors::no_operation,
520         class Ring,
521         typename IOType, typename InputType1, typename InputType2,
522         typename Coords, typename RIT, typename CIT, typename NIT,
523         Backend implementation = config::default_backend
524     >
525     RC mxv(
526         Vector< IOType, implementation, Coords > &u,
527         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
528         const Vector< InputType1, implementation, Coords > &v,
529         const Ring &ring,
530         typename std::enable_if<
531             grb::is_semiring< Ring >::value, void
532         >::type * = nullptr
533     ) {
534 #ifdef _DEBUG
535         std::cerr « "Selected backend does not implement grb::mxv\n";
536 #endif
537 #ifndef NDEBUG
538         const bool backend_does_not_support_mxv = false;
539         assert( backend_does_not_support_mxv );
540 #endif
541         (void) u;
542         (void) A;
543         (void) v;
544         (void) ring;
545         return UNSUPPORTED;
546     }
547
560     template<
561         Descriptor descr = descriptors::no_operation,
562         class Ring,
563         typename IOType, typename InputType1, typename InputType2,
564         typename InputType3,
565         typename Coords, typename RIT, typename CIT, typename NIT,
566         enum Backend implementation = config::default_backend
567     >
568     RC vxm(
569         Vector< IOType, implementation, Coords > &u,
570         const Vector< InputType3, implementation, Coords > &mask,
571         const Vector< InputType1, implementation, Coords > &v,
572         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
573         const Ring &ring = Ring(),
574         const Phase &phase = EXECUTE,
575         typename std::enable_if<
576             grb::is_semiring< Ring >::value, void
577         >::type * = nullptr
578     ) {
579 #ifdef _DEBUG
580         std::cerr « "Selected backend does not implement grb::vxm (output-masked)\n";
581 #endif
582 #ifndef NDEBUG
583         const bool selected_backend_does_not_support_output_masked_vxm = false;
584         assert( selected_backend_does_not_support_output_masked_vxm );
585 #endif
586         (void) u;
587         (void) mask;
588         (void) v;
589         (void) A;
590         (void) ring;
591         (void) phase;
592         return UNSUPPORTED;
```

```
593     }
594
607     template<
608         Descriptor descr = descriptors::no_operation,
609         class Ring,
610         typename IOType, typename InputType1, typename InputType2,
611         typename Coords, typename RIT, typename CIT, typename NIT,
612         enum Backend implementation = config::default_backend
613     >
614     RC vxm(
615         Vector< IOType, implementation, Coords > &u,
616         const Vector< InputType1, implementation, Coords > &v,
617         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
618         const Ring &ring = Ring(),
619         const Phase &phase = EXECUTE,
620         typename std::enable_if<
621             grb::is_semiring< Ring >::value, void
622         >::type * = nullptr
623     ) {
624 #ifdef _DEBUG
625         std::cerr << "Selected backend does not implement grb::vxm\n";
626 #endif
627 #ifndef NDEBUG
628         const bool selected_backend_does_not_support_vxm = false;
629         assert( selected_backend_does_not_support_vxm );
630 #endif
631         (void) u;
632         (void) v;
633         (void) A;
634         (void) ring;
635         return UNSUPPORTED;
636     }
637
651     template<
652         Descriptor descr = descriptors::no_operation,
653         class AdditiveMonoid, class MultiplicativeOperator,
654         typename IOType, typename InputType1, typename InputType2,
655         typename InputType3, typename InputType4,
656         typename Coords, typename RIT, typename CIT, typename NIT,
657         Backend backend
658     >
659     RC vxm(
660         Vector< IOType, backend, Coords > &u,
661         const Vector< InputType3, backend, Coords > &mask,
662         const Vector< InputType1, backend, Coords > &v,
663         const Vector< InputType4, backend, Coords > &v_mask,
664         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
665         const AdditiveMonoid &add = AdditiveMonoid(),
666         const MultiplicativeOperator &mul = MultiplicativeOperator(),
667         const Phase &phase = EXECUTE,
668         const typename std::enable_if<
669             grb::is_monoid< AdditiveMonoid >::value &&
670             grb::is_operator< MultiplicativeOperator >::value &&
671             !grb::is_object< IOType >::value &&
672             !grb::is_object< InputType1 >::value &&
673             !grb::is_object< InputType2 >::value &&
674             !grb::is_object< InputType3 >::value &&
675             !grb::is_object< InputType4 >::value &&
676             !std::is_same< InputType2, void >::value,
677         void >::type * const = nullptr
678     ) {
679 #ifdef _DEBUG
680         std::cerr << "Selected backend does not implement vxm (doubly-masked)\n";
681 #endif
682 #ifndef NDEBUG
683         const bool selected_backed_does_not_support_doubly_masked_vxm = false;
684         assert( selected_backed_does_not_support_doubly_masked_vxm );
685 #endif
686         (void) u;
687         (void) mask;
688         (void) v;
689         (void) v_mask;
690         (void) A;
691         (void) add;
692         (void) mul;
693         return UNSUPPORTED;
694     }
695
709     template<
710         Descriptor descr = descriptors::no_operation,
711         class AdditiveMonoid, class MultiplicativeOperator,
712         typename IOType, typename InputType1, typename InputType2,
713         typename InputType3, typename InputType4,
714         typename Coords, typename RIT, typename CIT, typename NIT,
715         Backend backend
716     >
717     RC mxv(
```

```
718         Vector< IOType, backend, Coords > &u,
719         const Vector< InputType3, backend, Coords > &mask,
720         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
721         const Vector< InputType1, backend, Coords > &v,
722         const Vector< InputType4, backend, Coords > &v_mask,
723         const AdditiveMonoid &add = AdditiveMonoid(),
724         const MultiplicativeOperator &mul = MultiplicativeOperator(),
725         const Phase &phase = EXECUTE,
726         const typename std::enable_if<
727             grb::is_monoid< AdditiveMonoid >::value &&
728             grb::is_operator< MultiplicativeOperator >::value &&
729             !grb::is_object< IOType >::value &&
730             !grb::is_object< InputType1 >::value &&
731             !grb::is_object< InputType2 >::value &&
732             !grb::is_object< InputType3 >::value &&
733             !grb::is_object< InputType4 >::value &&
734             !std::is_same< InputType2,
735         void >::value, void >::type * const = nullptr
736     ) {
737 #ifdef _DEBUG
738         std::cerr « "Selected backend does not implement mxv (doubly-masked)\n";
739 #endif
740 #ifndef NDEBUG
741         const bool selected_backed_does_not_support_doubly_masked_mxv = false;
742         assert( selected_backed_does_not_support_doubly_masked_mxv );
743 #endif
744         (void) u;
745         (void) mask;
746         (void) A;
747         (void) v;
748         (void) v_mask;
749         (void) add;
750         (void) mul;
751         return UNSUPPORTED;
752     }
753
767     template<
768         Descriptor descr = descriptors::no_operation,
769         class AdditiveMonoid, class MultiplicativeOperator,
770         typename IOType, typename InputType1, typename InputType2,
771         typename InputType3,
772         typename Coords, typename RIT, typename CIT, typename NIT,
773         Backend backend
774     >
775     RC mxv(
776         Vector< IOType, backend, Coords > &u,
777         const Vector< InputType3, backend, Coords > &mask,
778         const Matrix< InputType2, backend, RIT, NIT, CIT > &A,
779         const Vector< InputType1, backend, Coords > &v,
780         const AdditiveMonoid & add = AdditiveMonoid(),
781         const MultiplicativeOperator & mul = MultiplicativeOperator(),
782         const Phase &phase = EXECUTE,
783         const typename std::enable_if<
784             grb::is_monoid< AdditiveMonoid >::value &&
785             grb::is_operator< MultiplicativeOperator >::value &&
786             !grb::is_object< IOType >::value &&
787             !grb::is_object< InputType1 >::value &&
788             !grb::is_object< InputType2 >::value &&
789             !grb::is_object< InputType3 >::value &&
790             !std::is_same< InputType2, void >::value,
791         void >::type * const = nullptr
792     ) {
793 #ifdef _DEBUG
794         std::cerr « "Selected backend does not implement "
795             « "singly-masked monoid-op mxv\n";
796 #endif
797 #ifndef NDEBUG
798         const bool selected_backed_does_not_support_masked_monop_mxv = false;
799         assert( selected_backed_does_not_support_masked_monop_mxv );
800 #endif
801         (void) u;
802         (void) mask;
803         (void) A;
804         (void) v;
805         (void) add;
806         (void) mul;
807         return UNSUPPORTED;
808     }
809
823     template<
824         Descriptor descr = descriptors::no_operation,
825         class AdditiveMonoid, class MultiplicativeOperator,
826         typename IOType, typename InputType1, typename InputType2,
827         typename Coords, typename RIT, typename CIT, typename NIT,
828         Backend backend
829     >
830     RC vxm(
```

```
831         Vector< IOType, backend, Coords > &u,
832         const Vector< InputType1, backend, Coords > &v,
833         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
834         const AdditiveMonoid &add = AdditiveMonoid(),
835         const MultiplicativeOperator &mul = MultiplicativeOperator(),
836         const Phase &phase = EXECUTE,
837         const typename std::enable_if<
838             grb::is_monoid< AdditiveMonoid >::value &&
839             grb::is_operator< MultiplicativeOperator >::value &&
840             !grb::is_object< IOType >::value &&
841             !grb::is_object< InputType1 >::value &&
842             !grb::is_object< InputType2 >::value &&
843             !std::is_same< InputType2, void >::value,
844         void >::type * const = nullptr
845     ) {
846 #ifdef _DEBUG
847         std::cerr « "Selected backend does not implement vxm "
848             « "(unmasked, monoid-op version )\n";
849 #endif
850 #ifndef NDEBUG
851         const bool selected_backed_does_not_support_monop_vxm = false;
852         assert( selected_backed_does_not_support_monop_vxm );
853 #endif
854         (void) u;
855         (void) v;
856         (void) A;
857         (void) add;
858         (void) mul;
859         return UNSUPPORTED;
860     }
861
875     template<
876         Descriptor descr = descriptors::no_operation,
877         class AdditiveMonoid, class MultiplicativeOperator,
878         typename IOType, typename InputType1, typename InputType2,
879         typename InputType3,
880         typename Coords, typename RIT, typename CIT, typename NIT,
881         Backend implementation
882     >
883     RC vxm(
884         Vector< IOType, implementation, Coords > &u,
885         const Vector< InputType3, implementation, Coords > &mask,
886         const Vector< InputType1, implementation, Coords > &v,
887         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
888         const AdditiveMonoid &add = AdditiveMonoid(),
889         const MultiplicativeOperator &mul = MultiplicativeOperator(),
890         const Phase &phase = EXECUTE,
891         typename std::enable_if<
892             grb::is_monoid< AdditiveMonoid >::value &&
893             grb::is_operator< MultiplicativeOperator >::value &&
894             !grb::is_object< IOType >::value &&
895             !grb::is_object< InputType1 >::value &&
896             !grb::is_object< InputType2 >::value &&
897             !std::is_same< InputType2, void >::value,
898         void >::type * = nullptr
899     ) {
900 #ifdef _DEBUG
901         std::cerr « "Selected backend does not implement grb::vxm (output-masked)\n";
902 #endif
903 #ifndef NDEBUG
904         const bool selected_backed_does_not_support_masked_monop_vxm = false;
905         assert( selected_backed_does_not_support_masked_monop_vxm );
906 #endif
907         (void) u;
908         (void) mask;
909         (void) v;
910         (void) A;
911         (void) add;
912         (void) mul;
913         return UNSUPPORTED;
914     }
915
929     template<
930         Descriptor descr = descriptors::no_operation,
931         class AdditiveMonoid, class MultiplicativeOperator,
932         typename IOType, typename InputType1, typename InputType2,
933         typename Coords, typename RIT, typename CIT, typename NIT,
934         Backend backend
935     >
936     RC mxv(
937         Vector< IOType, backend, Coords > &u,
938         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
939         const Vector< InputType1, backend, Coords > &v,
940         const AdditiveMonoid &add = AdditiveMonoid(),
941         const MultiplicativeOperator &mul = MultiplicativeOperator(),
942         const Phase &phase = EXECUTE,
943         const typename std::enable_if<
```

```
944             grb::is_monoid< AdditiveMonoid >::value &&
945             grb::is_operator< MultiplicativeOperator >::value &&
946             !grb::is_object< IOType >::value &&
947             !grb::is_object< InputType1 >::value &&
948             !grb::is_object< InputType2 >::value &&
949             !std::is_same< InputType2, void >::value,
950         void >::type * const = nullptr
951     ) {
952 #ifdef _DEBUG
953         std::cerr « "Selected backend does not implement grb::mxv (unmasked)\n";
954 #endif
955 #ifndef NDEBUG
956         const bool selected_backed_does_not_support_monop_mxv = false;
957         assert( selected_backed_does_not_support_monop_mxv );
958 #endif
959         (void) u;
960         (void) A;
961         (void) v;
962         (void) add;
963         (void) mul;
964         return UNSUPPORTED;
965     }
966
969 } // namespace grb
970
971 #endif // end _H_GRB_BLAS2_BASE
972
```

10.39 blas3.hpp File Reference

Defines the ALP/GraphBLAS level-3 API.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Functions

- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType1 , typename InputType2 , typename CIT , typename RIT , typename NIT , class Semiring , Backend backend>
  RC mxm (Matrix< OutputType, backend, CIT, RIT, NIT > &C, const Matrix< InputType1, backend, CIT, RIT, NIT > &A, const Matrix< InputType2, backend, CIT, RIT, NIT > &B, const Semiring &ring=Semiring(), const Phase &phase=EXECUTE)

  Unmasked and in-place sparse matrix–sparse matrix multiplication (SpMSpM), $C+ = A + B$.
- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , Backend backend, typename Coords >
  RC zip (Matrix< OutputType, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Vector< InputType3, backend, Coords > &z, const Phase &phase=EXECUTE)

  The grb::zip merges three vectors into a matrix.
- template<Descriptor descr = descriptors::no_operation, typename InputType1 , typename InputType2 , typename InputType3 , typename RIT , typename CIT , typename NIT , Backend backend, typename Coords >
  RC zip (Matrix< void, backend, RIT, CIT, NIT > &A, const Vector< InputType1, backend, Coords > &x, const Vector< InputType2, backend, Coords > &y, const Phase &phase=EXECUTE)

  Merges two vectors into a `void` *matrix.*

10.39.1 Detailed Description

Defines the ALP/GraphBLAS level-3 API.

Author

A. N. Yzelman

10.40 blas3.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
26 #ifndef _H_GRB_BLAS3_BASE
27 #define _H_GRB_BLAS3_BASE
28
29 #include <graphblas/backends.hpp>
30 #include <graphblas/phase.hpp>
31
32 #include "matrix.hpp"
33 #include "vector.hpp"
34
35
36 namespace grb {
37
87     template<
88         Descriptor descr = descriptors::no_operation,
89         typename OutputType, typename InputType1, typename InputType2,
90         typename CIT, typename RIT, typename NIT,
91         class Semiring,
92         Backend backend
93     >
94     RC mxm(
95         Matrix< OutputType, backend, CIT, RIT, NIT > &C,
96         const Matrix< InputType1, backend, CIT, RIT, NIT > &A,
97         const Matrix< InputType2, backend, CIT, RIT, NIT > &B,
98         const Semiring &ring = Semiring(),
99         const Phase &phase = EXECUTE
100     ) {
101 #ifdef _DEBUG
102         std::cerr « "Selected backend does not implement grb::mxm "
103             « "(semiring version)\n";
104 #endif
105 #ifndef NDEBUG
106         const bool selected_backend_does_not_support_mxm = false;
107         assert( selected_backend_does_not_support_mxm );
108 #endif
109         (void) C;
110         (void) A;
111         (void) B;
112         (void) ring;
113         (void) phase;
114         // this is the generic stub implementation
115         return UNSUPPORTED;
116     }
117
175     template<
176         Descriptor descr = descriptors::no_operation,
177         typename OutputType, typename InputType1, typename InputType2,
178         typename InputType3, typename RIT, typename CIT, typename NIT,
179         Backend backend, typename Coords
180     >
```

```
181     RC zip(
182         Matrix< OutputType, backend, RIT, CIT, NIT > &A,
183         const Vector< InputType1, backend, Coords > &x,
184         const Vector< InputType2, backend, Coords > &y,
185         const Vector< InputType3, backend, Coords > &z,
186         const Phase &phase = EXECUTE
187     ) {
188         (void) x;
189         (void) y;
190         (void) z;
191         (void) phase;
192 #ifdef _DEBUG
193         std::cerr « "Selected backend does not implement grb::zip (vectors into "
194             « "matrices, non-void)\n";
195 #endif
196 #ifndef NDEBUG
197         const bool selected_backend_does_not_support_zip_from_vectors_to_matrix
198             = false;
199         assert( selected_backend_does_not_support_zip_from_vectors_to_matrix );
200 #endif
201         const RC ret = grb::clear( A );
202         return ret == SUCCESS ? UNSUPPORTED : ret;
203     }
204
217     template<
218         Descriptor descr = descriptors::no_operation,
219         typename InputType1, typename InputType2, typename InputType3,
220         typename RIT, typename CIT, typename NIT,
221         Backend backend, typename Coords
222     >
223     RC zip(
224         Matrix< void, backend, RIT, CIT, NIT > &A,
225         const Vector< InputType1, backend, Coords > &x,
226         const Vector< InputType2, backend, Coords > &y,
227         const Phase &phase = EXECUTE
228     ) {
229         (void) x;
230         (void) y;
231         (void) phase;
232 #ifdef _DEBUG
233         std::cerr « "Selected backend does not implement grb::zip (vectors into "
234             « "matrices, void)\n";
235 #endif
236 #ifndef NDEBUG
237         const bool selected_backend_does_not_support_zip_from_vectors_to_void_matrix
238             = false;
239         assert( selected_backend_does_not_support_zip_from_vectors_to_void_matrix );
240 #endif
241         const RC ret = grb::clear( A );
242         return ret == SUCCESS ? UNSUPPORTED : ret;
243     }
244
249 } // namespace grb
250
251 #endif // end _H_GRB_BLAS3_BASE
252
```

10.41 collectives.hpp File Reference

Specifies some basic collectives which may be used within a multi-process ALP program.

Classes

- class collectives< implementation >

  A static class defining various collective operations on scalars.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.41.1 Detailed Description

Specifies some basic collectives which may be used within a multi-process ALP program.

Author

A. N. Yzelman & J. M. Nash

Date

20th of February, 2017

10.42 collectives.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
28 #ifndef _H_GRB_COLL_BASE
29 #define _H_GRB_COLL_BASE
30
31 #include <graphblas/backends.hpp>
32 #include <graphblas/descriptors.hpp>
33 #include <graphblas/rc.hpp>
34
35
36 namespace grb {
37
45     template< enum Backend implementation >
46     class collectives {
47
48         private:
49
51             collectives() {}
52
53
54         public:
55
116             template<
117                 Descriptor descr = descriptors::no_operation,
118                 typename Operator,
119                 typename IOType
120             >
121             static RC allreduce( IOType &inout, const Operator op = Operator() ) {
122                 (void) inout;
123                 (void)op;
124                 return PANIC;
125             }
126
186             template<
187                 Descriptor descr = descriptors::no_operation,
188                 typename Operator,
189                 typename IOType
190             >
191             static RC reduce(
192                 IOType &inout,
193                 const size_t root = 0,
194                 const Operator op = Operator()
195             ) {
196                 (void) inout;
```

```
197                 (void) op;
198                 (void) root;
199                 return PANIC;
200             }
201
243             template< typename IOType >
244             static RC broadcast( IOType &inout, const size_t root = 0 ) {
245                 (void) inout;
246                 (void) root;
247                 return PANIC;
248             }
249
256             template< Descriptor descr = descriptors::no_operation, typename IOType >
257             static RC broadcast(
258                 IOType * inout,
259                 const size_t size,
260                 const size_t root = 0
261             ) {
262                 (void) inout;
263                 (void) size;
264                 (void) root;
265                 return PANIC;
266             }
267
268     }; // end class "collectives"
269
270 } // end namespace grb
271
272 #endif // end _H_GRB_COLL_BASE
273
```

10.43 config.hpp File Reference

Defines both configuration parameters effective for all backends, as well as defines structured ways of passing backend-specific parameters.

Classes

- class BENCHMARKING

  Benchmarking default configuration parameters.
- class CACHE_LINE_SIZE

  Contains information about the target architecture cache line size.
- class IMPLEMENTATION< backend >

  Collects a series of implementation choices corresponding to some given backend.
- class MEMORY

  Memory configuration parameters.
- class SIMD_SIZE

  The SIMD size, in bytes.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::config

  Compile-time configuration constants as well as implementation details that are derived from such settings.

Typedefs

- typedef unsigned int ColIndexType

  What data type should be used to store column indices.
- typedef size_t NonzeroIndexType

  What data type should be used to refer to an array containing nonzeroes.
- typedef unsigned int RowIndexType

  What data type should be used to store row indices.
- typedef unsigned int VectorIndexType

  What data type should be used to store vector indices.

10.43.1 Detailed Description

Defines both configuration parameters effective for all backends, as well as defines structured ways of passing backend-specific parameters.

Author

A. N. Yzelman

Date

8th of August, 2016

10.44 base/config.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
28 #ifndef _H_GRB_CONFIG_BASE
29 #define _H_GRB_CONFIG_BASE
30
31 #include <cstddef> //size_t
32 #include <string>
33
34 #include <assert.h>
35 #include <unistd.h> //sysconf
36
37 #include <graphblas/backends.hpp>
38
39 #ifndef _GRB_NO_STDIO
40  #include <iostream> //std::cout
41 #endif
42
43 // if the user did not define _GRB_BACKEND, set it to the default sequential
44 // implementation
45 #ifndef _GRB_BACKEND
46  #define _GRB_BACKEND reference
47 #endif
48
```

```
49
50 namespace grb {
51
56     namespace config {
57
79         static constexpr grb::Backend default_backend = _GRB_BACKEND;
80
86         class CACHE_LINE_SIZE {
87
88             private:
89
94                 static constexpr size_t bytes = 64;
95
96             public:
97
104                 static constexpr size_t value() {
105                     return bytes;
106                 }
107
108         };
109
115         class SIMD_SIZE {
116
117             private:
118
123                 static constexpr size_t bytes = 64;
124
125             public:
126
133                 static constexpr size_t value() {
134                     return bytes;
135                 }
136
137         };
138
145         template< typename T >
146         class SIMD_BLOCKSIZE {
147
148             public:
149
158                 static constexpr size_t unsafe_value() {
159                     return SIMD_SIZE::value() / sizeof( T );
160                 }
161
168                 static constexpr size_t value() {
169                     return unsafe_value() > 0 ? unsafe_value() : 1;
170                 }
171
172         };
173
185         class HARDWARE_THREADS {
186
187             public:
188
200                 static long value() {
201                     return sysconf( _SC_NPROCESSORS_ONLN );
202                 }
203
204         };
205
211         class BENCHMARKING {
212
213             public:
214
216                 static constexpr size_t inner() {
217                     return 1;
218                 }
219
221                 static constexpr size_t outer() {
222                     return 10;
223                 }
224
225         };
226
232         class MEMORY {
233
234             public:
235
237                 static constexpr size_t l1_cache_size() {
238                     return 32768;
239                 }
240
244                 static constexpr size_t big_memory() {
245                     return 31;
246                 } // 2GB
247
269                 static constexpr double random_access_memspeed() {
```

```
270                     return 147.298;
271                 }
272
294                 static constexpr double stream_memspeed() {
295                     return 1931.264;
296                 }
297
305                 static bool report(
306                     const std::string prefix, const std::string action,
307                     const size_t size, const bool printNewline = true
308                 ) {
309 #ifdef _GRB_NO_STDIO
310                     (void) prefix;
311                     (void) action;
312                     (void) size;
313                     (void) printNewline;
314                     return false;
315 #else
316                     constexpr size_t big =
317  #ifdef _DEBUG
318                         true;
319  #else
320                         ( 1ul << big_memory() );
321  #endif
322                     if( size >= big ) {
323                         std::cout << "Info: ";
324                         std::cout << prefix << " ";
325                         std::cout << action << " ";
326                         if( sizeof( size_t ) * 8 > 40 && ( size >> 40 ) > 2 ) {
327                             std::cout << ( size >> 40 ) << " TB of memory";
328                         } else if( sizeof( size_t ) * 8 > 30 && ( size >> 30 ) > 2 ) {
329                             std::cout << ( size >> 30 ) << " GB of memory";
330                         } else if( sizeof( size_t ) * 8 > 20 && ( size >> 20 ) > 2 ) {
331                             std::cout << ( size >> 20 ) << " MB of memory";
332                         } else if( sizeof( size_t ) * 8 > 10 && ( size >> 10 ) > 2 ) {
333                             std::cout << ( size >> 10 ) << " kB of memory";
334                         } else {
335                             std::cout << size << " bytes of memory";
336                         }
337                         if( printNewline ) {
338                             std::cout << ".\n";
339                         }
340                         return true;
341                     }
342                     return false;
343 #endif
344                 }
345         };
346
386         template< grb::Backend backend = default_backend >
387         class IMPLEMENTATION {
388 #ifdef __DOXYGEN__
389             public:
390
397                 static constexpr ALLOC_MODE defaultAllocMode();
398
405                 static constexpr ALLOC_MODE sharedAllocMode();
406
421                 static constexpr bool fixedVectorCapacities();
422 #endif
423         };
424
436         typedef unsigned int RowIndexType;
437
449         typedef unsigned int ColIndexType;
450
462         typedef size_t NonzeroIndexType;
463
475         typedef unsigned int VectorIndexType;
476
477     } // namespace config
478
479 } // namespace grb
480
481 #endif // end _H_GRB_CONFIG_BASE
482
```

10.45 config.hpp File Reference

Contains the configuration parameters for the BSP1D backend.

Classes

- class IMPLEMENTATION< BSP1D >

  This class collects configuration parameters that are specific to the grb::BSP1D and grb::hybrid backends.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::config

  Compile-time configuration constants as well as implementation details that are derived from such settings.

10.45.1 Detailed Description

Contains the configuration parameters for the BSP1D backend.

Author

A. N. Yzelman

Date

5th of May, 2017

10.46 bsp1d/config.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_BSP1D_CONFIG
28 #define _H_GRB_BSP1D_CONFIG
29
30 #include <assert.h>
31
32 #include <graphblas/reference/config.hpp>
33
34 // if not defined, we set the backend of the BSP1D implementation to the
35 // reference implementation
36 #ifndef _GRB_BSP1D_BACKEND
37  #pragma message "_GRB_BSP1D_BACKEND was not set-- auto-selecting reference"
38  #define _GRB_BSP1D_BACKEND reference
39 #endif
40
41
42 namespace grb {
43
44     namespace config {
45
```

```
64          template<>
65          class IMPLEMENTATION< BSP1D > {
66
67              private:
68
74                  static bool set;
75
81                  static grb::config::ALLOC_MODE mode;
82
88                  static void deduce() noexcept;
89
90
91              public:
92
96                  static constexpr ALLOC_MODE defaultAllocMode() {
97                      return grb::config::ALLOC_MODE::ALIGNED;
98                  }
99
106                  static constexpr bool fixedVectorCapacities() {
107                      return IMPLEMENTATION< _GRB_BSP1D_BACKEND >::fixedVectorCapacities();
108                  }
109
127                  static grb::config::ALLOC_MODE sharedAllocMode() noexcept;
128
134                  static constexpr Backend coordinatesBackend() {
135                      return IMPLEMENTATION< _GRB_BSP1D_BACKEND >::coordinatesBackend();
136                  }
137
138          };
139
142      } // namespace config
143
144 } // namespace grb
145
146 #endif // end "_H_GRB_BSP1D_CONFIG"
147
```

10.47 config.hpp File Reference

Configuration settings for the nonblocking backend.

Classes

- class ANALYTIC_MODEL

  Configuration parameters relating to the analytic model employed by the nonblocking backend.
- class IMPLEMENTATION< nonblocking >

  Implementation-dependent configuration parameters for the nonblocking backend.
- class PIPELINE

  Configuration parameters relating to the pipeline data structure.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::config

  Compile-time configuration constants as well as implementation details that are derived from such settings.

10.47.1 Detailed Description

Configuration settings for the nonblocking backend.

Author

Aristeidis Mastoras

Date

16th of May, 2022

10.48 nonblocking/config.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_NONBLOCKING_CONFIG
28 #define _H_GRB_NONBLOCKING_CONFIG
29
30 #include <graphblas/base/config.hpp>
31 #include <graphblas/reference/config.hpp>
32
33
34 namespace grb {
35
46     namespace config {
47
53         class PIPELINE {
54
55             public:
56
64                 static constexpr const size_t max_pipelines = 4;
65
76                 static constexpr const size_t max_containers = 16;
77
85                 static constexpr const size_t max_depth = 16;
86
95                 static constexpr const size_t max_tiles = 1 « 16;
96
101                  static constexpr const bool warn_if_exceeded = true;
102
107                  static constexpr const bool warn_if_not_native = true;
108
109         };
110
117         class ANALYTIC_MODEL {
118
119             public:
120
127                 static constexpr const size_t MIN_TILE_SIZE = 512;
128
134                 static constexpr const double L1_CACHE_USAGE_PERCENTAGE = 0.98;
135
136         };
137
153         template<>
154         class IMPLEMENTATION< nonblocking > {
155
```

```
156             public:
157
162                 static constexpr ALLOC_MODE defaultAllocMode() {
163                     return ALLOC_MODE::ALIGNED;
164                 }
165
170                 static constexpr ALLOC_MODE sharedAllocMode() {
171                     return ALLOC_MODE::INTERLEAVED;
172                 }
173
182                 static constexpr Backend coordinatesBackend() {
183                     return nonblocking;
184                 }
185
192                 static constexpr bool fixedVectorCapacities() {
193                     return true;
194                 }
195
196         };
197
198     } // namespace config
199
202 } // namespace grb
203
204 #endif // end "_H_GRB_NONBLOCKING_CONFIG"
205
```

10.49 config.hpp File Reference

Contains the configuration parameters for the reference and reference_omp backends.

Classes

- class IMPLEMENTATION< reference >

  This class collects configuration parameters that are specific to the grb::reference backend.
- class IMPLEMENTATION< reference_omp >

  This class collects configuration parameters that are specific to the grb::reference_omp backend.
- class PREFETCHING< backend >

  Default prefetching settings for reference and reference_omp backends.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::config

  Compile-time configuration constants as well as implementation details that are derived from such settings.

Enumerations

- enum ALLOC_MODE { ALIGNED , INTERLEAVED }

  The memory allocation modes implemented in the grb::reference and the grb::reference_omp backends.

Functions

- std::string **toString** (const ALLOC_MODE mode)

  Converts instances of grb::config::ALLOC_MODE to a descriptive lower-case string.

10.49.1 Detailed Description

Contains the configuration parameters for the reference and reference_omp backends.

Author

A. N. Yzelman

Date

13th of September 2017.

10.50 reference/config.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
28 #ifndef _H_GRB_REFERENCE_CONFIG
29 #define _H_GRB_REFERENCE_CONFIG
30
31 #include <graphblas/base/config.hpp>
32
33
34 namespace grb {
35
36     namespace config {
37
54         enum ALLOC_MODE {
55
57             ALIGNED,
58
60             INTERLEAVED
61
62         };
63
70         std::string toString( const ALLOC_MODE mode );
71
96         template< Backend backend >
97         class PREFETCHING {
98
99             // guard against unintended use
100             static_assert( backend == reference || backend == reference_omp,
101                 "Instantiating for non-reference backend" );
102
103             public:
104
108                 static constexpr bool enabled() {
109                     return false;
110                 }
111
117                 static constexpr size_t distance() {
118                     return 128;
119                 }
120
121         };
122
139         template<>
140         class IMPLEMENTATION< reference > {
141
```

```
142             public:
143
145                 static constexpr ALLOC_MODE defaultAllocMode() {
146                     return ALLOC_MODE::ALIGNED;
147                 }
148
150                 static constexpr ALLOC_MODE sharedAllocMode() {
151                     return ALLOC_MODE::ALIGNED;
152                 }
153
160                 static constexpr bool fixedVectorCapacities() {
161                     return true;
162                 }
163
172                 static constexpr size_t vectorBufferSize( const size_t, const size_t ) {
173                     return 0;
174                 }
175
181                 static constexpr Backend coordinatesBackend() {
182                     return reference;
183                 }
184
185         };
186
203         template<>
204         class IMPLEMENTATION< reference_omp > {
205
206             private:
207
214                 static constexpr size_t minVectorBufferChunksize() {
215                     return CACHE_LINE_SIZE::value();
216                 }
217
232                 static constexpr size_t absVectorBufferSize() {
233                     return 0;
234                 }
235
252                 static constexpr double relVectorBufferSize() {
253                     return 1;
254                 }
255
256
257         public:
258
263                 static constexpr ALLOC_MODE defaultAllocMode() {
264                     return ALLOC_MODE::ALIGNED;
265                 }
266
271                 static constexpr ALLOC_MODE sharedAllocMode() {
272                     // return ALLOC_MODE::ALIGNED; //DBG
273                     return ALLOC_MODE::INTERLEAVED;
274                 }
275
281                 static constexpr Backend coordinatesBackend() {
282                     return reference_omp;
283                 }
284
291                 static constexpr bool fixedVectorCapacities() {
292                     return true;
293                 }
294
311                 static inline size_t vectorBufferSize( const size_t n, const size_t T ) {
312                     size_t ret;
313                     if( absVectorBufferSize() > 0 ) {
314                         (void)n;
315                         ret = absVectorBufferSize();
316                     } else {
317                         constexpr const double factor = relVectorBufferSize();
318                         static_assert( factor > 0, "Configuration error" );
319                         ret = factor * n;
320                     }
321                     ret = std::max( ret, T * minVectorBufferChunksize() );
322                     ret += T;
323                     if( ret % T > 0 ) {
324                         ret += T - ( ret % T );
325                     }
326                     ret = std::max( 2 * T, ret );
327                     assert( ret % T == 0 );
328                     return ret;
329                 }
330
331         };
332
335     } // namespace config
336
337 } // namespace grb
338
```

```
339 #endif // end "_H_GRB_REFERENCE_CONFIG"
340
```

10.51 exec.hpp File Reference

Specifies the grb::Launcher functionalities.

Classes

- class Launcher< mode, backend >

  A group of user processes that together execute ALP programs.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Enumerations

- enum EXEC_MODE { AUTOMATIC = 0 , MANUAL , FROM_MPI }

  The various ways in which the grb::Launcher can be used to execute an ALP program.

10.51.1 Detailed Description

Specifies the grb::Launcher functionalities.

Author

A. N. Yzelman

Date

17th of April, 2017

10.52 exec.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_EXEC_BASE
28 #define _H_GRB_EXEC_BASE
29
30 #include <stdexcept>
31 #include <string>
32
33 #include <graphblas/backends.hpp>
34 #include <graphblas/rc.hpp>
35
36 #ifndef _GRB_NO_STDIO
37  #include <iostream>
38 #endif
39
40
41 namespace grb {
42
56     enum EXEC_MODE {
57
62         AUTOMATIC = 0,
63
69         MANUAL,
70
76         FROM_MPI
77
78     };
79
96     template< enum EXEC_MODE mode, enum Backend backend >
97     class Launcher {
98
99         public :
100
140             Launcher(
141                 const size_t process_id = 0,
142                 const size_t nprocs = 1,
143                 const std::string hostname = "localhost",
144                 const std::string port = "0"
145             ) {
146                 // spec does not specify any constrants on hostname and port
147                 // so accept (and ignore) anything
148                 (void) hostname; (void) port;
149
150 #ifndef _GRB_NO_EXCEPTIONS
151                 // sanity checks on process_id and nprocs
152                 if( nprocs == 0 ) {
153                     throw std::invalid_argument( "Total number of user processes must be "
154                         "strictly larger than zero." );
155                 }
156                 if( process_id >= nprocs ) {
157                     throw std::invalid_argument( "Process ID must be strictly smaller than "
158                         "total number of user processes." );
159                 }
160 #endif
161             }
162
209             template< typename T, typename U >
210             RC exec(
211                 void ( *alp_program )( const T &, U & ),
212                 const T &data_in,
213                 U &data_out,
214                 const bool broadcast = false
215             ) const {
216                 (void) alp_program;
217                 (void) data_in;
218                 (void) data_out;
219                 (void) broadcast;
```

```
220                 // stub implementation, should be overridden by specialised backend,
221                 // so return error code
222                 return PANIC;
223             }
224 
251             template< typename U >
252             RC exec(
253                 void ( *alp_program )( const void *, const size_t, U & ),
254                 const void * data_in,
255                 const size_t in_size,
256                 U &data_out,
257                 const bool broadcast = false
258             ) const {
259                 (void) alp_program;
260                 (void) data_in;
261                 (void) in_size;
262                 (void) data_out;
263                 (void) broadcast;
264                 return PANIC;
265             }
266 
322             static RC finalize() {
323                 return PANIC;
324             }
325 
326     }; // end class 'Launcher'
327 
328 } // end namespace "grb"
329 
330 #endif // end _H_GRB_EXEC_BASE
331 
```

10.53 init.hpp File Reference

Specifies the grb::init and grb::finalize functionalities.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Functions

- template<enum Backend backend = config::default_backend>
  RC finalize ()

  Finalises an ALP/GraphBLAS context opened by the last call to grb::init.
- template<enum Backend backend = config::default_backend>
  RC init ()

  Initialises the calling user process.
- template<enum Backend backend = config::default_backend>
  RC init (const size_t s, const size_t P, void *const implementation_data)

  Initialises the calling user process.

10.53.1 Detailed Description

Specifies the grb::init and grb::finalize functionalities.

Author

A. N. Yzelman

Date

24th of January, 2017

10.54 init.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_INIT_BASE
28 #define _H_GRB_INIT_BASE
29
30 #include <graphblas/rc.hpp>
31
32 #include "config.hpp"
33
34
35 namespace grb {
36
144     template< enum Backend backend = config::default_backend >
145     RC init( const size_t s, const size_t P, void * const implementation_data ) {
146         (void) s;
147         (void) P;
148         (void) implementation_data;
149         return PANIC;
150     }
151
171     template< enum Backend backend = config::default_backend >
172     RC init() {
173         return grb::init< backend >( 0, 1, nullptr );
174     }
175
209     template< enum Backend backend = config::default_backend >
210     RC finalize() {
211         return PANIC;
212     }
213
214 } // namespace grb
215
216 #endif // end _H_GRB_INIT_BASE
217
```

10.55 io.hpp File Reference

Specifies all I/O primitives for use with ALP/GraphBLAS containers.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Functions

- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename fwd_iterator3 = const InputType * __restrict__, Backend implementation = config::default_backend>
  RC buildMatrixUnique (Matrix< InputType, implementation > &A, fwd_iterator1 I, const fwd_iterator1 I_end, fwd_iterator2 J, const fwd_iterator2 J_end, fwd_iterator3 V, const fwd_iterator3 V_end, const IOMode mode)

Assigns nonzeroes to the matrix from a coordinate format.

- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename fwd_iterator3 = const InputType * __restrict__, Backend implementation = config::default_backend>
  RC **buildMatrixUnique** (Matrix< InputType, implementation > &A, fwd_iterator1 I, fwd_iterator2 J, fwd_↩ iterator3 V, const size_t nz, const IOMode mode)

  Alias that transforms a set of pointers and an array length to the buildMatrixUnique variant based on iterators.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename RIT , typename CIT , typename NIT , typename fwd_iterator , Backend implementation = config::default_backend>
  RC buildMatrixUnique (Matrix< InputType, implementation, RIT, CIT, NIT > &A, fwd_iterator start, const fwd_iterator end, const IOMode mode)

  Version of buildMatrixUnique that works by supplying a single iterator (instead of three).
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename RIT , typename CIT , typename NIT , typename fwd_iterator1 = const size_t * __restrict__, typename fwd_iterator2 = const size_t * __restrict__, typename length_type = size_↩ t, Backend implementation = config::default_backend>
  RC **buildMatrixUnique** (Matrix< InputType, implementation, RIT, CIT, NIT > &A, fwd_iterator1 I, fwd_↩ iterator2 J, const length_type nz, const IOMode mode)

  Version of the above buildMatrixUnique that handles nullptr value pointers.
- template<Descriptor descr = descriptors::no_operation, typename InputType , typename fwd_iterator , Backend backend, typename Coords >
  RC buildVector (Vector< InputType, backend, Coords > &x, fwd_iterator start, const fwd_iterator end, const IOMode mode)

  Constructs a dense vector from a container of exactly grb::size(x) elements.
- template<Descriptor descr = descriptors::no_operation, typename InputType , class Merger = operators::right_assign< InputType >, typename fwd_iterator1 , typename fwd_iterator2 , Backend backend, typename Coords >
  RC buildVector (Vector< InputType, backend, Coords > &x, fwd_iterator1 ind_start, const fwd_↩ iterator1 ind_end, fwd_iterator2 val_start, const fwd_iterator2 val_end, const IOMode mode, const Merger &merger=Merger())

  Ingests possibly sparse input from a container to which iterators are provided.
- template<Descriptor descr = descriptors::no_operation, typename InputType , class Merger = operators::right_assign< InputType >, typename fwd_iterator1 , typename fwd_iterator2 , Backend backend, typename Coords >
  RC buildVectorUnique (Vector< InputType, backend, Coords > &x, fwd_iterator1 ind_start, const fwd_↩ iterator1 ind_end, fwd_iterator2 val_start, const fwd_iterator2 val_end, const IOMode mode)

  Ingests a set of nonzeroes into a given vector x.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t capacity (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Queries the capacity of the given ALP/GraphBLAS container.
- template<typename InputType , Backend backend, typename Coords >
  size_t capacity (const Vector< InputType, backend, Coords > &x) noexcept

  Queries the capacity of the given ALP/GraphBLAS container.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  RC clear (Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Clears a given matrix of all nonzeroes.
- template<typename DataType , Backend backend, typename Coords >
  RC clear (Vector< DataType, backend, Coords > &x) noexcept

  Clears a given vector of all nonzeroes.
- template<typename ElementType , typename RIT , typename CIT , typename NIT , Backend implementation = config::default_↩ backend>
  uintptr_t getID (const Matrix< ElementType, implementation, RIT, CIT, NIT > &x)

  Specialisation of getID for matrix containers.
- template<typename ElementType , typename Coords , Backend implementation = config::default_backend>
  uintptr_t getID (const Vector< ElementType, implementation, Coords > &x)

  Function that returns a unique ID for a given non-empty container.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t ncols (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

Requests the column size of a given matrix.

- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t nnz (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Retrieve the number of nonzeroes contained in this matrix.
- template<typename DataType , Backend backend, typename Coords >
  size_t nnz (const Vector< DataType, backend, Coords > &x) noexcept

  Request the number of nonzeroes in a given vector.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  size_t nrows (const Matrix< InputType, backend, RIT, CIT, NIT > &A) noexcept

  Requests the row size of a given matrix.
- template<typename InputType , Backend backend, typename RIT , typename CIT , typename NIT >
  RC resize (Matrix< InputType, backend, RIT, CIT, NIT > &A, const size_t new_nz) noexcept

  Resizes the nonzero capacity of this matrix.
- template<typename InputType , Backend backend, typename Coords >
  RC resize (Vector< InputType, backend, Coords > &x, const size_t new_nz) noexcept

  Resizes the nonzero capacity of this vector.
- template<Descriptor descr = descriptors::no_operation, typename DataType , typename T , typename Coords , Backend backend>
  RC set (Vector< DataType, backend, Coords > &x, const T val, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr) noexcept

  Sets all elements of a vector to the given value.
- template<Descriptor descr = descriptors::no_operation, typename DataType , typename MaskType , typename T , Backend backend, typename Coords >
  RC set (Vector< DataType, reference, Coords > &x, const Vector< MaskType, backend, Coords > &mask, const T val, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr)

  Sets all elements of a vector to the given value whenever the given mask evaluates `true`.
- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename InputType , Backend backend, typename Coords >
  RC set (Vector< OutputType, backend, Coords > &x, const Vector< InputType, backend, Coords > &y, const Phase &phase=EXECUTE)

  Sets the content of a given vector x to be equal to that of another given vector y.
- template<Descriptor descr = descriptors::no_operation, typename OutputType , typename MaskType , typename InputType , Backend backend, typename Coords >
  RC set (Vector< OutputType, backend, Coords > &x, const Vector< MaskType, backend, Coords > &mask, const Vector< InputType, backend, Coords > &y, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< OutputType >::value &&!grb::is_object< MaskType >::value &&!grb::is_object< InputType >::value, void >::type *const =nullptr)

  Sets the content of a given vector x to be equal to that of another given vector y.
- template<Descriptor descr = descriptors::no_operation, typename DataType , typename T , Backend backend, typename Coords >
  RC setElement (Vector< DataType, backend, Coords > &x, const T val, const size_t i, const Phase &phase=EXECUTE, const typename std::enable_if< !grb::is_object< DataType >::value &&!grb::is_object< T >::value, void >::type *const =nullptr)

  Sets the element of a given vector at a given position to a given value.
- template<typename DataType , Backend backend, typename Coords >
  size_t size (const Vector< DataType, backend, Coords > &x) noexcept

  Request the size of a given vector.
- template<Backend backend = config::default_backend>
  RC wait ()

  Depending on the backend, ALP/GraphBLAS primitives may be non-blocking, meaning that the operation immediately returns even though the requested computation has not been performed.
- template<Backend backend, typename InputType , typename RIT , typename CIT , typename NIT , typename... Args>
  RC wait (const Matrix< InputType, backend, RIT, CIT, NIT > &A, const Args &... args)

  A variant of grb::wait that executes, at minimum, all nonblocking primitives required for computing a given output matrix as well as, optionally, for any additional output containers given in the variadic argument list.

- template<Backend backend, typename InputType , typename Coords , typename... Args>
  RC wait (const Vector< InputType, backend, Coords > &x, const Args &... args)

  A variant of grb::wait that executes, at minimum, all nonblocking primitives required for computing a given output vector as well as, optionally, for any additional output containers given in the variadic argument list.

10.55.1 Detailed Description

Specifies all I/O primitives for use with ALP/GraphBLAS containers.

Author

A. N. Yzelman

Date

21st of February, 2017

10.56 io.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #include <type_traits>
28 #include <typeinfo>
29
30 #ifndef _H_GRB_IO_BASE
31 #define _H_GRB_IO_BASE
32
33 #include <graphblas/rc.hpp>
34 #include <graphblas/phase.hpp>
35 #include <graphblas/iomode.hpp>
36 #include <graphblas/SynchronizedNonzeroIterator.hpp>
37 #include <graphblas/utils/iterators/type_traits.hpp>
38
39 #include "matrix.hpp"
40 #include "vector.hpp"
41
42 #include <assert.h>
43
44
45 namespace grb {
46
153     template<
154         typename ElementType, typename Coords,
155         Backend implementation = config::default_backend
156     >
157     uintptr_t getID(
158         const Vector< ElementType, implementation, Coords > &x
159     ) {
160         (void) x;
161 #ifndef NDEBUG
162         const bool this_is_an_invalid_default_implementation = false;
163 #endif
164         assert( this_is_an_invalid_default_implementation );
165         return static_cast< uintptr_t >(-1);
```

```
166     }
167
174     template<
175         typename ElementType, typename RIT, typename CIT, typename NIT,
176         Backend implementation = config::default_backend
177     >
178     uintptr_t getID(
179         const Matrix< ElementType, implementation, RIT, CIT, NIT > &x
180     ) {
181         (void) x;
182 #ifndef NDEBUG
183         const bool this_is_an_invalid_default_implementation = false;
184 #endif
185         assert( this_is_an_invalid_default_implementation );
186         return static_cast< uintptr_t >(-1);
187     }
188
231     template<
232         typename DataType,
233         Backend backend, typename Coords
234     >
235     size_t size( const Vector< DataType, backend, Coords > &x ) noexcept {
236 #ifndef NDEBUG
237         const bool may_not_call_base_size = false;
238 #endif
239         (void) x;
240         assert( may_not_call_base_size );
241         return SIZE_MAX;
242     }
243
282     template<
283         typename InputType, Backend backend,
284         typename RIT, typename CIT, typename NIT
285     >
286     size_t nrows(
287         const Matrix< InputType, backend, RIT, CIT, NIT > &A
288     ) noexcept {
289 #ifndef NDEBUG
290         const bool may_not_call_base_nrows = false;
291 #endif
292         (void) A;
293         assert( may_not_call_base_nrows );
294         return SIZE_MAX;
295     }
296
335     template<
336         typename InputType, Backend backend,
337         typename RIT, typename CIT, typename NIT
338     >
339     size_t ncols(
340         const Matrix< InputType, backend, RIT, CIT, NIT > &A
341     ) noexcept {
342 #ifndef NDEBUG
343         const bool may_not_call_base_ncols = false;
344 #endif
345         (void) A;
346         assert( may_not_call_base_ncols );
347         return SIZE_MAX;
348     }
349
385     template<
386         typename InputType, Backend backend, typename Coords
387     >
388     size_t capacity( const Vector< InputType, backend, Coords > &x ) noexcept {
389 #ifndef NDEBUG
390         const bool should_not_call_base_vector_capacity = false;
391 #endif
392         assert( should_not_call_base_vector_capacity );
393         (void) x;
394         return SIZE_MAX;
395     }
396
428     template<
429         typename InputType, Backend backend,
430         typename RIT, typename CIT, typename NIT
431     >
432     size_t capacity(
433         const Matrix< InputType, backend, RIT, CIT, NIT > &A
434     ) noexcept {
435 #ifndef NDEBUG
436         const bool should_not_call_base_matrix_capacity = false;
437 #endif
438         assert( should_not_call_base_matrix_capacity );
439         (void) A;
440         return SIZE_MAX;
441     }
442
```

```
478     template< typename DataType, Backend backend, typename Coords >
479     size_t nnz( const Vector< DataType, backend, Coords > &x ) noexcept {
480 #ifndef NDEBUG
481         const bool should_not_call_base_vector_nnz = false;
482 #endif
483         (void) x;
484         assert( should_not_call_base_vector_nnz );
485         return SIZE_MAX;
486     }
487
519     template<
520         typename InputType, Backend backend,
521         typename RIT, typename CIT, typename NIT
522     >
523     size_t nnz(
524         const Matrix< InputType, backend, RIT, CIT, NIT > &A
525     ) noexcept {
526 #ifndef NDEBUG
527         const bool should_not_call_base_matrix_nnz = false;
528 #endif
529         (void) A;
530         assert( should_not_call_base_matrix_nnz );
531         return SIZE_MAX;
532     }
533
573     template< typename DataType, Backend backend, typename Coords >
574     RC clear( Vector< DataType, backend, Coords > &x ) noexcept {
575 #ifndef NDEBUG
576         const bool should_not_call_base_vector_clear = false;
577 #endif
578         (void) x;
579         assert( should_not_call_base_vector_clear );
580         return UNSUPPORTED;
581     }
582
618     template<
619         typename InputType, Backend backend,
620         typename RIT, typename CIT, typename NIT
621     >
622     RC clear(
623         Matrix< InputType, backend, RIT, CIT, NIT > &A
624     ) noexcept {
625 #ifndef NDEBUG
626         const bool should_not_call_base_matrix_clear = false;
627 #endif
628         (void) A;
629         assert( should_not_call_base_matrix_clear );
630         return UNSUPPORTED;
631     }
632
699     template<
700         typename InputType,
701         Backend backend, typename Coords
702     >
703     RC resize(
704         Vector< InputType, backend, Coords > &x,
705         const size_t new_nz
706     ) noexcept {
707 #ifndef NDEBUG
708         const bool should_not_call_base_vector_resize = false;
709 #endif
710         (void) x;
711         (void) new_nz;
712         assert( should_not_call_base_vector_resize );
713         return UNSUPPORTED;
714     }
715
782     template<
783         typename InputType, Backend backend,
784         typename RIT, typename CIT, typename NIT
785     >
786     RC resize(
787         Matrix< InputType, backend, RIT, CIT, NIT > &A, const size_t new_nz
788     ) noexcept {
789 #ifndef NDEBUG
790         const bool should_not_call_base_matrix_resize = false;
791 #endif
792         (void) A;
793         (void) new_nz;
794         assert( should_not_call_base_matrix_resize );
795         return UNSUPPORTED;
796     }
797
852     template<
853         Descriptor descr = descriptors::no_operation,
854         typename DataType, typename T,
855         typename Coords, Backend backend
```

```
856     >
857     RC set(
858         Vector< DataType, backend, Coords > &x, const T val,
859         const Phase &phase = EXECUTE,
860         const typename std::enable_if<
861             !grb::is_object< DataType >::value &&
862             !grb::is_object< T >::value,
863         void >::type * const = nullptr
864     ) noexcept {
865 #ifndef NDEBUG
866         const bool should_not_call_base_vector_set = false;
867         assert( should_not_call_base_vector_set );
868 #endif
869         (void) x;
870         (void) val;
871         (void) phase;
872         return UNSUPPORTED;
873     }
874
936     template<
937         Descriptor descr = descriptors::no_operation,
938         typename DataType, typename MaskType, typename T,
939         Backend backend, typename Coords
940     >
941     RC set(
942         Vector< DataType, reference, Coords > &x,
943         const Vector< MaskType, backend, Coords > &mask,
944         const T val,
945         const Phase &phase = EXECUTE,
946         const typename std::enable_if<
947             !grb::is_object< DataType >::value && !grb::is_object< T >::value,
948         void >::type * const = nullptr
949     ) {
950 #ifndef NDEBUG
951         const bool should_not_call_base_masked_vector_set = false;
952         assert( should_not_call_base_masked_vector_set );
953 #endif
954         (void) x;
955         (void) mask;
956         (void) val;
957         (void) phase;
958         return UNSUPPORTED;
959     }
960
993     template<
994         Descriptor descr = descriptors::no_operation,
995         typename OutputType, typename InputType,
996         Backend backend, typename Coords
997     >
998     RC set(
999         Vector< OutputType, backend, Coords > &x,
1000          const Vector< InputType, backend, Coords > &y,
1001          const Phase &phase = EXECUTE
1002      ) {
1003 #ifndef NDEBUG
1004          const bool should_not_call_base_vector_set_copy = false;
1005          assert( should_not_call_base_vector_set_copy );
1006 #endif
1007          (void) x;
1008          (void) y;
1009          (void) phase;
1010          return UNSUPPORTED;
1011      }
1012
1054     template<
1055         Descriptor descr = descriptors::no_operation,
1056         typename OutputType, typename MaskType, typename InputType,
1057         Backend backend, typename Coords
1058     >
1059     RC set(
1060         Vector< OutputType, backend, Coords > &x,
1061         const Vector< MaskType, backend, Coords > &mask,
1062         const Vector< InputType, backend, Coords > &y,
1063         const Phase &phase = EXECUTE,
1064         const typename std::enable_if< !grb::is_object< OutputType >::value &&
1065             !grb::is_object< MaskType >::value &&
1066             !grb::is_object< InputType >::value,
1067         void >::type * const = nullptr
1068     ) {
1069 #ifndef NDEBUG
1070         const bool should_not_call_base_vector_set_copy_masked = false;
1071         assert( should_not_call_base_vector_set_copy_masked );
1072 #endif
1073         (void) x;
1074         (void) mask;
1075         (void) y;
1076         (void) phase;
```

```
1077         return UNSUPPORTED;
1078     }
1079
1123     template<
1124         Descriptor descr = descriptors::no_operation,
1125         typename DataType, typename T,
1126         Backend backend, typename Coords
1127     >
1128     RC setElement(
1129         Vector< DataType, backend, Coords > &x,
1130         const T val,
1131         const size_t i,
1132         const Phase &phase = EXECUTE,
1133         const typename std::enable_if< !grb::is_object< DataType >::value &&
1134             !grb::is_object< T >::value, void >::type * const = nullptr
1135     ) {
1136 #ifndef NDEBUG
1137         const bool should_not_call_base_setElement = false;
1138         assert( should_not_call_base_setElement );
1139 #endif
1140         (void) x;
1141         (void) val;
1142         (void) i;
1143         (void) phase;
1144         return UNSUPPORTED;
1145     }
1146
1153     template<
1154         Descriptor descr = descriptors::no_operation,
1155         typename InputType, typename fwd_iterator,
1156         Backend backend, typename Coords
1157     >
1158     RC buildVector(
1159         Vector< InputType, backend, Coords > &x,
1160         fwd_iterator start, const fwd_iterator end,
1161         const IOMode mode
1162     ) {
1163         operators::right_assign< InputType > accum;
1164         return buildVector< descr >( x, accum, start, end, mode );
1165     }
1166
1174     template< Descriptor descr = descriptors::no_operation,
1175         typename InputType,
1176         class Merger = operators::right_assign< InputType >,
1177         typename fwd_iterator1, typename fwd_iterator2,
1178         Backend backend, typename Coords
1179     >
1180     RC buildVector( Vector< InputType, backend, Coords > &x,
1181         fwd_iterator1 ind_start, const fwd_iterator1 ind_end,
1182         fwd_iterator2 val_start, const fwd_iterator2 val_end,
1183         const IOMode mode, const Merger & merger = Merger()
1184     ) {
1185         operators::right_assign< InputType > accum;
1186         return buildVector< descr >( x, accum, ind_start, ind_end, val_start, val_end,
1187             mode, merger );
1188     }
1189
1221     template<
1222         Descriptor descr = descriptors::no_operation,
1223         typename InputType,
1224         class Merger = operators::right_assign< InputType >,
1225         typename fwd_iterator1, typename fwd_iterator2,
1226         Backend backend, typename Coords
1227     >
1228     RC buildVectorUnique(
1229         Vector< InputType, backend, Coords > &x,
1230         fwd_iterator1 ind_start, const fwd_iterator1 ind_end,
1231         fwd_iterator2 val_start, const fwd_iterator2 val_end,
1232         const IOMode mode
1233     ) {
1234         return buildVector< descr | descriptors::no_duplicates >(
1235             x,
1236             ind_start, ind_end,
1237             val_start, val_end,
1238             mode
1239         );
1240     }
1241
1328     template<
1329         Descriptor descr = descriptors::no_operation,
1330         typename InputType,
1331         typename fwd_iterator1 = const size_t * __restrict__,
1332         typename fwd_iterator2 = const size_t * __restrict__,
1333         typename fwd_iterator3 = const InputType * __restrict__,
1334         Backend implementation = config::default_backend
1335     >
1336     RC buildMatrixUnique(
```

```
1337         Matrix< InputType, implementation > &A,
1338         fwd_iterator1 I, const fwd_iterator1 I_end,
1339         fwd_iterator2 J, const fwd_iterator2 J_end,
1340         fwd_iterator3 V, const fwd_iterator3 V_end,
1341         const IOMode mode
1342     ) {
1343         // derive synchronized iterator
1344         auto start = internal::makeSynchronized(
1345             I, J, V,
1346             I_end, J_end, V_end
1347         );
1348         const auto end = internal::makeSynchronized(
1349             I_end, J_end, V_end,
1350             I_end, J_end, V_end
1351         );
1352         // defer to other signature
1353         return buildMatrixUnique< descr >( A, start, end, mode );
1354     }
1355
1360     template<
1361         Descriptor descr = descriptors::no_operation,
1362         typename InputType,
1363         typename fwd_iterator1 = const size_t * __restrict__,
1364         typename fwd_iterator2 = const size_t * __restrict__,
1365         typename fwd_iterator3 = const InputType * __restrict__,
1366         Backend implementation = config::default_backend
1367     >
1368     RC buildMatrixUnique(
1369         Matrix< InputType, implementation > &A,
1370         fwd_iterator1 I, fwd_iterator2 J, fwd_iterator3 V,
1371         const size_t nz, const IOMode mode
1372     ) {
1373         return buildMatrixUnique< descr >(
1374             A,
1375             I, I + nz,
1376             J, J + nz,
1377             V, V + nz,
1378             mode
1379         );
1380     }
1381
1386     template<
1387         Descriptor descr = descriptors::no_operation,
1388         typename InputType, typename RIT, typename CIT, typename NIT,
1389         typename fwd_iterator1 = const size_t * __restrict__,
1390         typename fwd_iterator2 = const size_t * __restrict__,
1391         typename length_type = size_t,
1392         Backend implementation = config::default_backend
1393     >
1394     RC buildMatrixUnique(
1395         Matrix< InputType, implementation, RIT, CIT, NIT > &A,
1396         fwd_iterator1 I, fwd_iterator2 J,
1397         const length_type nz, const IOMode mode
1398     ) {
1399         // derive synchronized iterator
1400         auto start = internal::makeSynchronized( I, J, I + nz, J + nz );
1401         const auto end = internal::makeSynchronized(
1402             I + nz, J + nz, I + nz, J + nz );
1403         // defer to other signature
1404         return buildMatrixUnique< descr >( A, start, end, mode );
1405     }
1406
1448     template<
1449         Descriptor descr = descriptors::no_operation,
1450         typename InputType, typename RIT, typename CIT, typename NIT,
1451         typename fwd_iterator,
1452         Backend implementation = config::default_backend
1453     >
1454     RC buildMatrixUnique(
1455         Matrix< InputType, implementation, RIT, CIT, NIT > &A,
1456         fwd_iterator start, const fwd_iterator end,
1457         const IOMode mode
1458     ) {
1459         (void) A;
1460         (void) start;
1461         (void) end;
1462         (void) mode;
1463 #ifndef NDEBUG
1464         std::cerr « "Should not call base grb::buildMatrixUnique" « std::endl;
1465         const bool should_not_call_base_buildMatrixUnique = false;
1466         assert( should_not_call_base_buildMatrixUnique );
1467 #endif
1468         return PANIC;
1469     }
1470
1517     template< Backend backend = config::default_backend >
1518     RC wait() {
```

```
1519 #ifndef NDEBUG
1520         const bool should_not_call_base_wait = false;
1521         assert( should_not_call_base_wait );
1522 #endif
1523         return UNSUPPORTED;
1524     }
1525
1551     template<
1552         Backend backend, typename InputType, typename Coords,
1553         typename... Args
1554     >
1555     RC wait(
1556         const Vector< InputType, backend, Coords > &x,
1557         const Args &... args
1558     ) {
1559 #ifndef NDEBUG
1560         const bool should_not_call_base_vector_wait = false;
1561         assert( should_not_call_base_vector_wait );
1562 #endif
1563         (void) x;
1564         return wait( args... );
1565     }
1566
1592     template<
1593         Backend backend,
1594         typename InputType, typename RIT, typename CIT, typename NIT,
1595         typename... Args
1596     >
1597     RC wait(
1598         const Matrix< InputType, backend, RIT, CIT, NIT > &A,
1599         const Args &... args
1600     ) {
1601 #ifndef NDEBUG
1602         const bool should_not_call_base_matrix_wait = false;
1603         assert( should_not_call_base_matrix_wait );
1604 #endif
1605         (void) A;
1606         return wait( args... );
1607     }
1608
1611 } // namespace grb
1612
1613 #endif // end _H_GRB_IO_BASE
1614
```

10.57 matrix.hpp File Reference

Specifies the ALP/GraphBLAS matrix container.

Classes

- class Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType >::const_iterator

  A standard iterator for an ALP/GraphBLAS matrix.
- class Matrix< D, implementation, RowIndexType, ColIndexType, NonzeroIndexType >

  An ALP/GraphBLAS matrix.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.57.1 Detailed Description

Specifies the ALP/GraphBLAS matrix container.

Author

A. N. Yzelman

Date

10th of August, 2016

10.58 matrix.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_MATRIX_BASE
28 #define _H_GRB_MATRIX_BASE
29
30 #include <iterator>
31
32 #include <stddef.h>
33
34 #include <utility>
35
36 #include <graphblas/backends.hpp>
37 #include <graphblas/descriptors.hpp>
38 #include <graphblas/ops.hpp>
39 #include <graphblas/rc.hpp>
40
41
42 namespace grb {
43
65     template<
66         typename D, enum Backend implementation,
67         typename RowIndexType,
68         typename ColIndexType,
69         typename NonzeroIndexType
70     >
71     class Matrix {
72
73         public :
74
76             typedef Matrix<
77                 D, implementation,
78                 RowIndexType, ColIndexType, NonzeroIndexType
79             > self_type;
80
116              class const_iterator : public std::iterator<
117                  std::forward_iterator_tag,
118                  std::pair< std::pair< const size_t, const size_t >, const D >,
119                  size_t
120              > {
121
122                  public :
123
130                      bool operator==( const const_iterator &other ) const {
131                          (void)other;
```

```
132                         return false;
133                     }
134
138                     bool operator!=( const const_iterator &other ) const {
139                         (void)other;
140                         return true;
141                     }
142
155                     std::pair< const size_t, const D > operator*() const {
156                         return std::pair< const size_t, const D >();
157                     }
158
172                     const_iterator & operator++() {
173                         return *this;
174                     }
175
176             };
177
179             typedef D value_type;
180
205             Matrix( const size_t rows, const size_t columns, const size_t nz ) {
206                 (void) rows;
207                 (void) columns;
208                 (void) nz;
209             }
210
226             Matrix( const size_t rows, const size_t columns ) {
227                 (void)rows;
228                 (void)columns;
229             }
230
260             Matrix(
261                 const Matrix<
262                     D, implementation,
263                     RowIndexType, ColIndexType, NonzeroIndexType
264                 > &other
265             ) {
266                 (void) other;
267             }
268
283             Matrix( self_type &&other ) {
284                 (void) other;
285             }
286
303             self_type& operator=( self_type &&other ) noexcept {
304                 *this = std::move( other );
305                 return *this;
306             }
307
321             ~Matrix() {}
322
349             const_iterator cbegin() const {}
350
358             const_iterator begin() const {}
359
380             const_iterator cend() const {}
381
389             const_iterator end() const {}
390
391     };
392
393 } // end namespace "grb"
394
395 #endif // end _H_GRB_MATRIX_BASE
396
```

10.59 pinnedvector.hpp File Reference

Contains the specification for grb::PinnedVector.

Classes

- class PinnedVector< IOType, implementation >

  Provides a mechanism to access ALP containers from outside of an ALP context.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.59.1 Detailed Description

Contains the specification for grb::PinnedVector.

Author

A. N. Yzelman

10.60 pinnedvector.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
26 #ifndef _H_GRB_BASE_PINNEDVECTOR
27 #define _H_GRB_BASE_PINNEDVECTOR
28
29 #include <limits>
30
31 #include <graphblas/backends.hpp>
32 #include <graphblas/iomode.hpp>
33
34 #include "vector.hpp"
35
36
37 namespace grb {
38
74     template< typename IOType, enum Backend implementation >
75     class PinnedVector {
76
77         private :
78
82             static const constexpr bool
83                 function_was_not_implemented_in_the_selected_backend = false;
84
85
86         public :
87
133             template< typename Coord >
134             PinnedVector(
135                 const Vector< IOType, implementation, Coord > &vector,
136                 const IOMode mode
137             ) {
138                 (void) vector;
139                 (void) mode;
140                 assert( function_was_not_implemented_in_the_selected_backend );
141             }
142
153             PinnedVector() {
154                 assert( function_was_not_implemented_in_the_selected_backend );
155             }
156
166             ~PinnedVector() {
```

```
167                 assert( function_was_not_implemented_in_the_selected_backend );
168             }
169
180             inline size_t size() const noexcept {
181                 assert( function_was_not_implemented_in_the_selected_backend );
182                 return 0;
183             }
184
195             inline size_t nonzeroes() const noexcept {
196                 assert( function_was_not_implemented_in_the_selected_backend );
197                 return 0;
198             }
199
232             template< typename OutputType >
233             inline OutputType getNonzeroValue(
234                 const size_t k, const OutputType one = OutputType()
235             ) const noexcept {
236                 (void) k;
237                 assert( function_was_not_implemented_in_the_selected_backend );
238                 return one;
239             }
240
256             inline IOType getNonzeroValue(
257                 const size_t k
258             ) const noexcept {
259                 IOType ret;
260                 (void) k;
261                 assert( function_was_not_implemented_in_the_selected_backend );
262                 return ret;
263             }
264
282             inline size_t getNonzeroIndex(
283                 const size_t k
284             ) const noexcept {
285                 (void) k;
286                 assert( function_was_not_implemented_in_the_selected_backend );
287                 return std::numeric_limits< size_t >::max();
288             }
289
290
291     }; // end class grb::PinnedVector
292
293 } // end namespace "grb"
294
295 #endif // end _H_GRB_BASE_PINNEDVECTOR
296
```

10.61 properties.hpp File Reference

Provides a mechanism for inspecting properties of various backends.

Classes

- class Properties< backend >

  Collection of various properties on the given ALP/GraphBLAS backend.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.61.1 Detailed Description

Provides a mechanism for inspecting properties of various backends.

Author

A. N. Yzelman

Date

5th of May 2017

10.62 properties.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_PROPERTIES_BASE
28 #define _H_GRB_PROPERTIES_BASE
29
30 #include <graphblas/backends.hpp>
31
32
33 namespace grb {
34
51     template< enum Backend backend >
52     class Properties {
53
54 #ifdef __DOXYGEN__
55
56         public:
57
72             static constexpr const bool writableCaptured = true;
73
84             static constexpr const bool isBlockingExecution = true;
85
99             static constexpr const bool isNonblockingExecution = !isBlockingExecution;
100 #endif
101         };
102
103 } // namespace grb
104
105 #endif // end _H_GRB_PROPERTIES_BASE
106
```

10.63 spmd.hpp File Reference

Exposes facilities for direct SPMD programming.

Classes

- class spmd< implementation >

  For backends that support multiple user processes this class defines some basic primitives to support SPMD programming.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.63.1 Detailed Description

Exposes facilities for direct SPMD programming.

Author

A. N. Yzelman

Date

28th of April, 2017

10.64 spmd.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_BASE_SPMD
28 #define _H_GRB_BASE_SPMD
29
30 #include <cstddef> //size_t
31
32 #include <stdint.h> // SIZE_MAX
33
34 #include <graphblas/backends.hpp>
35 #include <graphblas/rc.hpp>
36
37 #include "config.hpp"
38
39
40 namespace grb {
41
50     template< Backend implementation >
51     class spmd {
52
53         public:
54
56             static inline size_t nprocs() noexcept {
```

```
57                  return 0;
58              }
59
61              static inline size_t pid() noexcept {
62                  return SIZE_MAX;
63              }
64
91              static enum RC sync( const size_t msgs_in = 0, const size_t msgs_out = 0 ) noexcept {
92                  (void) msgs_in;
93                  (void) msgs_out;
94                  return PANIC;
95              }
96
97      }; // end class "spmd"
98
99 } // namespace grb
100
101 #endif // end _H_GRB_BASE_SPMD
102
```

10.65 vector.hpp File Reference

Specifies the ALP/GraphBLAS vector container.

Classes

- class Vector< D, implementation, C >::const_iterator

  A standard iterator for the Vector< D > class.
- class Vector< D, implementation, C >

  A GraphBLAS vector.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.65.1 Detailed Description

Specifies the ALP/GraphBLAS vector container.

Author

A. N. Yzelman

Date

10th of August, 2016

10.66 vector.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_VECTOR_BASE
28 #define _H_GRB_VECTOR_BASE
29
30 #include <cstdlib>  //size_t
31 #include <iterator> //std::iterator
32 #include <stdexcept>
33 #include <utility> //pair
34
35 #include <graphblas/backends.hpp>
36 #include <graphblas/descriptors.hpp>
37 #include <graphblas/ops.hpp>
38 #include <graphblas/rc.hpp>
39
40
41 namespace grb {
42
63     template< typename D, enum Backend implementation, typename C >
64     class Vector {
65
66         public :
67
69             typedef D value_type;
70
124             typedef D& lambda_reference;
125
140             class const_iterator :
141                 public std::iterator<
142                     std::forward_iterator_tag,
143                     std::pair< const size_t, const D >,
144                     size_t
145                 >
146             {
147
148                 public :
149
151                     bool operator==( const const_iterator &other ) const {
152                         (void) other;
153                         return false;
154                     }
155
157                     bool operator!=( const const_iterator &other ) const {
158                         (void) other;
159                         return true;
160                     }
161
173                     std::pair< const size_t, const D > operator*() const {
174                         return std::pair< const size_t, const D >();
175                     }
176
190                     const_iterator & operator++() {
191                         return *this;
192                     }
193
194             };
195
226             Vector( const size_t n, const size_t nz ) {
227                 (void) n;
228                 (void) nz;
229             }
230
235             Vector( const size_t n ) {
236                 (void) n;
237             }
238
259             Vector( Vector< D, implementation, C > &&x ) noexcept {
```

```
260             (void) x;
261         }
262
278         Vector< D, implementation, C >& operator=(
279             Vector< D, implementation, C > &&x
280         ) noexcept {
281             (void) x;
282             return *this;
283         }
284
307         ~Vector() {}
308
310
340         const_iterator cbegin() const {
341             const_iterator ret;
342             return ret;
343         }
344
351         const_iterator begin() const {
352             const_iterator ret;
353             return ret;
354         }
355
357
359
380         const_iterator cend() const {
381             const_iterator ret;
382             return ret;
383         }
384
391         const_iterator end() const {
392             const_iterator ret;
393             return ret;
394         }
396
480         template<
481             Descriptor descr = descriptors::no_operation,
482             class Accum = typename operators::right_assign< D, D, D >,
483             typename fwd_iterator = const D * __restrict__
484         >
485         RC build(
486             const Accum &accum,
487             const fwd_iterator start, const fwd_iterator end,
488             fwd_iterator npos
489         ) {
490             (void) accum;
491             (void) start;
492             (void) end;
493             (void) npos;
494             return PANIC;
495         }
496
595         template<
596             Descriptor descr = descriptors::no_operation,
597             class Accum = operators::right_assign< D, D, D >,
598             typename ind_iterator = const size_t * __restrict__,
599             typename nnz_iterator = const D * __restrict__,
600             class Dup = operators::right_assign< D, D, D >
601         >
602         RC build(
603             const Accum &accum,
604             const ind_iterator ind_start, const ind_iterator ind_end,
605             const nnz_iterator nnz_start, const nnz_iterator nnz_end,
606             const Dup &dup = Dup()
607         ) {
608             (void) accum;
609             (void) ind_start;
610             (void) ind_end;
611             (void) nnz_start;
612             (void) nnz_end;
613             (void) dup;
614             return PANIC;
615         }
616
720         template<
721             Descriptor descr = descriptors::no_operation,
722             typename mask_type,
723             class Accum,
724             typename ind_iterator = const size_t * __restrict__,
725             typename nnz_iterator = const D * __restrict__,
726             class Dup = operators::right_assign< D, typename nnz_iterator::value_type, D >
727         >
728         RC build(
729             const Vector< mask_type, implementation, C > &mask,
730             const Accum &accum,
731             const ind_iterator ind_start,
732             const ind_iterator ind_end,
```

```
733                 const nnz_iterator nnz_start,
734                 const nnz_iterator nnz_end,
735                 const Dup &dup = Dup()
736             ) {
737                 (void) mask;
738                 (void) accum;
739                 (void) ind_start;
740                 (void) ind_end;
741                 (void) nnz_start;
742                 (void) nnz_end;
743                 (void) dup;
744                 return PANIC;
745             }
746
771             template< typename T >
772             RC size( T &size ) const {
773                 (void) size;
774                 return PANIC;
775             }
776
801             template< typename T >
802             RC nnz( T &nnz ) const {
803                 (void) nnz;
804                 return PANIC;
805             }
806
866             template< class Monoid >
867             lambda_reference operator()(
868                 const size_t i, const Monoid &monoid = Monoid()
869             ) {
870                 (void) i;
871                 (void) monoid;
872                 return PANIC;
873             }
874
919             lambda_reference operator[]( const size_t i ) {
920                 (void) i;
921  #ifndef _GRB_NO_EXCEPTIONS
922                 throw std::runtime_error(
923                     "Requesting lambda reference of unimplemented Vector backend."
924                 );
925  #endif
926             }
927     };
928
929 } // end namespace "grb"
930
931 #endif // _H_GRB_VECTOR_BASE
932
```

10.67 blas0.hpp File Reference

Defines the ALP/GraphBLAS level-0 API.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Functions

- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType1 , typename InputType2 , typename OutputType >
  static enum RC apply (OutputType &out, const InputType1 &x, const InputType2 &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType1 >::value &&!grb::is_object< InputType2 >::value &&!grb::is_object< OutputType >::value, void >::type *=nullptr)

  Out-of-place application of the operator OP on two data elements.

- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType , typename IOType >
  static RC foldl (IOType &x, const InputType &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Application of the operator OP on two data elements.
- template<Descriptor descr = descriptors::no_operation, class OP , typename InputType , typename IOType >
  static RC foldr (const InputType &x, IOType &y, const OP &op=OP(), const typename std::enable_if< grb::is_operator< OP >::value &&!grb::is_object< InputType >::value &&!grb::is_object< IOType >::value, void >::type *=nullptr)

  Application of the operator OP on two data elements.

10.67.1 Detailed Description

Defines the ALP/GraphBLAS level-0 API.

Author

A. N. Yzelman

Date

5th of December 2016

10.68 blas0.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_BLAS0
28 #define _H_GRB_BLAS0
29
30 #include <functional>
31 #include <stdexcept>
32 #include <type_traits> //enable_if
33
34 #include "graphblas/descriptors.hpp"
35 #include "graphblas/rc.hpp"
36 #include "graphblas/type_traits.hpp"
37
38 #define NO_CAST_ASSERT( x, y, z )                                                \
39     static_assert( x,                                                            \
40         "\n\n"                                                                   \
41         "******************************************************************" \
42         "******************************************************************" \
43         "**********************\n"                                               \
44         "*     ERROR      | " y " " z ".\n"                                     \
45         "******************************************************************" \
46         "******************************************************************" \
47         "**********************\n"                                               \
48         "* Possible fix 1 | Remove no_casting from the template parameters in "    \
49         "this call to " y ".\n"                                                  \
```

```
50          "* Possible fix 2 | Provide a left-hand side input value of the same "     \
51          "type as the first domain of the given operator.\n"                        \
52          "* Possible fix 3 | Provide a right-hand side input value of the same "    \
53          "type as the second domain of the given operator.\n"                       \
54          "* Possible fix 4 | Provide an output value of the same type as the "      \
55          "third domain of the given operator.\n"                                    \
56          "* Note that in case of in-place operators the left-hand side input or "   \
57          "right-hand side input also play the role of the output value.\n"          \
58          "*********************************************************************" \
59          "*********************************************************************" \
60          "**********************\n" );
61
62
63 namespace grb {
64
174     template<
175         Descriptor descr = descriptors::no_operation,
176         class OP,
177         typename InputType1, typename InputType2, typename OutputType
178     >
179     static enum RC apply(
180         OutputType &out,
181         const InputType1 &x,
182         const InputType2 &y,
183         const OP &op = OP(),
184         const typename std::enable_if<
185             grb::is_operator< OP >::value &&
186             !grb::is_object< InputType1 >::value &&
187             !grb::is_object< InputType2 >::value &&
188             !grb::is_object< OutputType >::value,
189         void >::type * = nullptr
190     ) {
191         // static sanity check
192         NO_CAST_ASSERT( ( !( descr & descriptors::no_casting ) || (
193                 std::is_same< InputType1, typename OP::D1 >::value &&
194                 std::is_same< InputType2, typename OP::D2 >::value &&
195                 std::is_same< OutputType, typename OP::D3 >::value
196             ) ),
197             "grb::apply (BLAS level 0)",
198             "Argument value types do not match operator domains while no_casting "
199             "descriptor was set"
200         );
201
202         // call apply
203         const typename OP::D1 left = static_cast< typename OP::D1 >( x );
204         const typename OP::D2 right = static_cast< typename OP::D2 >( y );
205         typename OP::D3 output = static_cast< typename OP::D3 >( out );
206         op.apply( left, right, output );
207         out = static_cast< OutputType >( output );
208
209         // done
210         return SUCCESS;
211     }
212
283     template<
284         Descriptor descr = descriptors::no_operation,
285         class OP, typename InputType, typename IOType
286     >
287     static RC foldr(
288         const InputType &x,
289         IOType &y,
290         const OP &op = OP(),
291         const typename std::enable_if<
292             grb::is_operator< OP >::value &&
293             !grb::is_object< InputType >::value &&
294             !grb::is_object< IOType >::value, void
295         >::type * = nullptr
296     ) {
297         // static sanity check
298         NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) || (
299                 std::is_same< InputType, typename OP::D1 >::value &&
300                 std::is_same< IOType, typename OP::D2 >::value &&
301                 std::is_same< IOType, typename OP::D3 >::value
302             ) ), "grb::foldr (BLAS level 0)",
303             "Argument value types do not match operator domains while no_casting "
304             "descriptor was set" );
305
306         // call foldr
307         const typename OP::D1 left = static_cast< typename OP::D1 >( x );
308         typename OP::D3 right = static_cast< typename OP::D3 >( y );
309         op.foldr( left, right );
310         y = static_cast< IOType >( right );
311
312         // done
313         return SUCCESS;
314     }
315
```

```
386     template<
387         Descriptor descr = descriptors::no_operation,
388         class OP,
389         typename InputType, typename IOType
390     >
391     static RC foldl(
392         IOType &x,
393         const InputType &y,
394         const OP &op = OP(),
395         const typename std::enable_if< grb::is_operator< OP >::value &&
396             !grb::is_object< InputType >::value &&
397             !grb::is_object< IOType >::value, void
398         >::type * = nullptr
399     ) {
400         // static sanity check
401         NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) || (
402                 std::is_same< IOType, typename OP::D1 >::value &&
403                 std::is_same< InputType, typename OP::D2 >::value &&
404                 std::is_same< IOType, typename OP::D3 >::value
405             ) ), "grb::foldl (BLAS level 0)",
406             "Argument value types do not match operator domains while no_casting "
407             "descriptor was set" );
408
409         // call foldl
410         typename OP::D1 left = static_cast< typename OP::D1 >( x );
411         const typename OP::D3 right = static_cast< typename OP::D3 >( y );
412         op.foldl( left, right );
413         x = static_cast< IOType >( left );
414
415         // done
416         return SUCCESS;
417     }
418
421     namespace internal {
422
437         template<
438             grb::Descriptor descr,
439             typename OutputType, typename D,
440             typename Enabled = void
441         >
442         class ValueOrIndex;
443
444         /* Version where use_index is allowed. */
445         template< grb::Descriptor descr, typename OutputType, typename D >
446         class ValueOrIndex<
447             descr,
448             OutputType, D,
449             typename std::enable_if<
450                 std::is_arithmetic< OutputType >::value &&
451                 !std::is_same< D, void >::value
452             >::type
453         > {
454
455             private:
456
457                 static constexpr const bool use_index = descr & grb::descriptors::use_index;
458
459                 static_assert( use_index || std::is_convertible< D, OutputType >::value,
460                     "Cannot convert to the requested output type" );
461
462
463             public:
464
465                 static OutputType getFromArray(
466                     const D * __restrict__ const x,
467                     const std::function< size_t( size_t ) > &src_local_to_global,
468                     const size_t index
469                 ) noexcept {
470                     if( use_index ) {
471                         return static_cast< OutputType >( src_local_to_global( index ) );
472                     } else {
473                         return static_cast< OutputType >( x[ index ] );
474                     }
475                 }
476
477                 static OutputType getFromScalar( const D &x, const size_t index ) noexcept {
478                     if( use_index ) {
479                         return static_cast< OutputType >( index );
480                     } else {
481                         return static_cast< OutputType >( x );
482                     }
483                 }
484
485         };
486
487         /* Version where use_index is not allowed. */
488         template< grb::Descriptor descr, typename OutputType, typename D >
```

```
489         class ValueOrIndex<
490             descr,
491             OutputType, D,
492             typename std::enable_if<
493                 !std::is_arithmetic< OutputType >::value &&
494                 !std::is_same< OutputType, void >::value
495             >::type
496         > {
497
498             static_assert( !(descr & descriptors::use_index),
499                 "use_index descriptor given while output type is not numeric" );
500
501             static_assert( std::is_convertible< D, OutputType >::value,
502                 "Cannot convert input to the given output type" );
503
504             public:
505
506                 static OutputType getFromArray(
507                     const D * __restrict__ const x,
508                     const std::function< size_t( size_t ) > &,
509                     const size_t index
510                 ) noexcept {
511                     return static_cast< OutputType >( x[ index ] );
512                 }
513
514                 static OutputType getFromScalar(
515                     const D &x, const size_t
516                 ) noexcept {
517                     return static_cast< OutputType >( x );
518                 }
519
520         };
521
542         template<
543             bool identity_left,
544             typename OutputType, typename InputType,
545             template< typename > class Identity,
546             typename Enabled = void
547         >
548         class CopyOrApplyWithIdentity;
549
550         /* The cast-and-assign version */
551         template<
552             bool identity_left,
553             typename OutputType, typename InputType,
554             template< typename > class Identity
555         >
556         class CopyOrApplyWithIdentity<
557             identity_left,
558             OutputType, InputType,
559             Identity,
560             typename std::enable_if<
561                 std::is_convertible< InputType, OutputType >::value
562             >::type
563         > {
564
565             public:
566
567                 template< typename Operator >
568                 static void set( OutputType &out, const InputType &in, const Operator & ) {
569                     out = static_cast< OutputType >( in );
570                 }
571
572         };
573
574         /* The operator with identity version */
575         template<
576             bool identity_left,
577             typename OutputType, typename InputType,
578             template< typename > class Identity
579         >
580         class CopyOrApplyWithIdentity<
581             identity_left,
582             OutputType, InputType,
583             Identity,
584             typename std::enable_if<
585                 !std::is_convertible< InputType, OutputType >::value
586             >::type
587         > {
588
589             public:
590
591                 template< typename Operator >
592                 static void set(
593                     OutputType &out, const InputType &in, const Operator &op
594                 ) {
595                     const auto identity = identity_left ?
```

```
596                         Identity< typename Operator::D1 >::value() :
597                         Identity< typename Operator::D2 >::value();
598                     if( identity_left ) {
599                         (void) grb::apply( out, identity, in, op );
600                     } else {
601                         (void) grb::apply( out, in, identity, op );
602                     }
603                 }
604 
605         };
606 
607     } // namespace internal
608 
609 } // namespace grb
610 
611 #undef NO_CAST_ASSERT
612 
613 #endif // end "_H_GRB_BLAS0"
614 
```

10.69 descriptors.hpp File Reference

Defines all ALP/GraphBLAS descriptors.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::descriptors

  Collection of standard descriptors.

Typedefs

- typedef unsigned int Descriptor

  Descriptors indicate pre- or post-processing for some or all of the arguments to an ALP/GraphBLAS call.

Functions

- std::string toString (const Descriptor descr)

  Translates a descriptor into a string.

Variables

- static constexpr Descriptor add_identity = 32

  For any call to a matrix computation, the input matrix A is instead interpreted as $A + I$, with I the identity matrix of dimension matching A.
- static constexpr Descriptor dense = 16

  Indicates that all input and output vectors to an ALP/GraphBLAS primitive are structurally dense.
- static constexpr Descriptor explicit_zero = 512

  Computation shall proceed with zeros (according to the current semiring) propagating throughout the requested computation.
- static constexpr Descriptor **invert_mask** = 1

  Inverts the mask prior to applying it.
- static constexpr Descriptor no_casting = 256

Disallows the standard casting of input parameters to a compatible domain in case they did not match exactly.

- static constexpr Descriptor no_duplicates = 4

  For data ingestion methods, such as grb::buildVector or grb::buildMatrix, this descriptor indicates that the input shall not contain any duplicate entries.

- static constexpr Descriptor **no_operation** = 0

  Indicates no additional pre- or post-processing on any of the GraphBLAS function arguments.

- static constexpr Descriptor **safe_overlap** = 1024

  Indicates overlapping input and output vectors is intentional and safe, due to, for example, the use of masks.

- static constexpr Descriptor structural = 8

  Uses the structure of a mask vector only.

- static constexpr Descriptor structural_complement = structural | invert_mask

  Uses the structural complement of a mask vector.

- static constexpr Descriptor **transpose_left** = 2048

  For operations involving two matrices, transposes the left-hand side input matrix prior to applying it.

- static constexpr Descriptor **transpose_matrix** = 2

  Transposes the input matrix prior to applying it.

- static constexpr Descriptor **transpose_right** = 4096

  For operations involving two matrices, transposes the right-hand side input matrix prior to applying it.

- static constexpr Descriptor use_index = 64

  Instead of using input vector elements, use the index of those elements.

10.69.1 Detailed Description

Defines all ALP/GraphBLAS descriptors.

Author

A. N. Yzelman

Date

15 March, 2016

10.70 descriptors.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_DESCRIPTOR
28 #define _H_GRB_DESCRIPTOR
29
30 #include <string>
31
```

```
32
33 namespace grb {
34
54     typedef unsigned int Descriptor;
55
57     namespace descriptors {
58
63         static constexpr Descriptor no_operation = 0;
64
66         static constexpr Descriptor invert_mask = 1;
67
71         static constexpr Descriptor transpose_matrix = 2;
72
91         static constexpr Descriptor no_duplicates = 4;
92
103         static constexpr Descriptor structural = 8;
104
117         static constexpr Descriptor structural_complement = structural | invert_mask;
118
151         static constexpr Descriptor dense = 16;
152
159         static constexpr Descriptor add_identity = 32;
160
167         static constexpr Descriptor use_index = 64;
168
196         static constexpr Descriptor no_casting = 256;
197
207         static constexpr Descriptor explicit_zero = 512;
208
213         static constexpr Descriptor safe_overlap = 1024;
214
219         static constexpr Descriptor transpose_left = 2048;
220
225         static constexpr Descriptor transpose_right = 4096;
226
227         // Put internal, backend-specific descriptors last
228
229
239         static constexpr Descriptor force_row_major = 8192;
240
248         std::string toString( const Descriptor descr );
249
250     } // namespace descriptors
251
252     namespace internal {
253
255         static constexpr Descriptor MAX_DESCRIPTOR_VALUE = 16383;
256
257     } // namespace internal
258
259 } // namespace grb
260
261 #endif
262
```

10.71 identities.hpp File Reference

Provides a set of standard identities for use with ALP.

Classes

- class infinity< D >
  Standard identity for the minimum operator.
- class logical_false< D >
  Standard identitity for the logical or operator.
- class logical_true< D >
  Standard identity for the logical AND operator.
- class negative_infinity< D >
  Standard identity for the maximum operator.
- class one< D >
  Standard identity for numerical multiplication.
- class zero< D >
  Standard identity for numerical addition.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::identities

  Standard identities common to many operators.

10.71.1 Detailed Description

Provides a set of standard identities for use with ALP.

Author

A. N. Yzelman

Date

11th of August, 2016

10.72 identities.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_IDENTITIES
28 #define _H_GRB_IDENTITIES
29
30 #include <limits>
31
32
33 namespace grb {
34
53     namespace identities {
54
56         template< typename D >
57         class zero {
58             static_assert( std::is_convertible< int, D >::value, "Cannot form identity under the
    requested domain" );
59
60         public:
65             static constexpr D value() {
66                 return static_cast< D >( 0 );
67             }
68         };
69         template< typename K, typename V >
70         class zero< std::pair< K, V > > {
71         public:
72             static constexpr std::pair< K, V > value() {
73                 return std::make_pair( zero< K >::value(), zero< V >::value() );
74             }
75         };
76
78         template< typename D >
```

```
79          class one {
80              static_assert( std::is_convertible< int, D >::value, "Cannot form identity under the
    requested domain" );
81
82          public:
87              static constexpr D value() {
88                  return static_cast< D >( 1 );
89              }
90          };
91          template< typename K, typename V >
92          class one< std::pair< K, V > > {
93          public:
94              static constexpr std::pair< K, V > value() {
95                  return std::make_pair( one< K >::value(), one< V >::value() );
96              }
97          };
98
100         template< typename D >
101         class infinity {
102             static_assert( std::is_arithmetic< D >::value, "Cannot form identity under the requested
    domain" );
103
104         public:
110             static constexpr D value() {
111                 return std::numeric_limits< D >::has_infinity ? std::numeric_limits< D >::infinity() :
    std::numeric_limits< D >::max();
112             }
113         };
114         template< typename K, typename V >
115         class infinity< std::pair< K, V > > {
116         public:
117             static constexpr std::pair< K, V > value() {
118                 return std::make_pair( infinity< K >::value(), infinity< V >::value() );
119             }
120         };
121
123         template< typename D >
124         class negative_infinity {
125             static_assert( std::is_arithmetic< D >::value, "Cannot form identity under the requested
    domain" );
126
127         public:
133             static constexpr D value() {
134                 return std::numeric_limits< D >::min() == 0 ? 0 : ( std::numeric_limits< D
    >::has_infinity ? -std::numeric_limits< D >::infinity() : std::numeric_limits< D >::min() );
135             }
136         };
137         template< typename K, typename V >
138         class negative_infinity< std::pair< K, V > > {
139         public:
140             static constexpr std::pair< K, V > value() {
141                 return std::make_pair( negative_infinity< K >::value(), negative_infinity< V >::value()
    );
142             }
143         };
144
150         template< typename D >
151         class logical_false {
152             static_assert( std::is_convertible< bool, D >::value, "Cannot form identity under the
    requested domain" );
153
154         public:
160             static const constexpr D value() {
161                 return static_cast< D >( false );
162             }
163         };
164         template< typename K, typename V >
165         class logical_false< std::pair< K, V > > {
166         public:
167             static constexpr std::pair< K, V > value() {
168                 return std::make_pair( logical_false< K >::value(), logical_false< V >::value() );
169             }
170         };
171
177         template< typename D >
178         class logical_true {
179             static_assert( std::is_convertible< bool, D >::value, "Cannot form identity under the
    requested domain" );
180
181         public:
187             static constexpr D value() {
188                 return static_cast< D >( true );
189             }
190         };
191         template< typename K, typename V >
192         class logical_true< std::pair< K, V > > {
193         public:
```

```
194             static constexpr std::pair< K, V > value() {
195                 return std::make_pair( logical_true< K >::value(), logical_true< V >::value() );
196             }
197         };
198
199     } // namespace identities
200 } // namespace grb
201
202 #endif
203
```

10.73 pregel.hpp File Reference

This file defines a vertex-centric programming API called ALP/Pregel, which automatically translates to standard ALP/GraphBLAS primitives.

Classes

- class Pregel< MatrixEntryType >

  A Pregel run-time instance.
- struct PregelState

  The state of the vertex-center Pregel program that the user may interface with.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::interfaces

  The namespace for programming APIs that automatically translate to ALP/GraphBLAS.
- namespace grb::interfaces::config

  Contains configurations for programming models that are simulated on top of ALP/GraphBLAS.

Enumerations

- enum SparsificationStrategy { NONE = 0 , ALWAYS , WHEN_REDUCED , WHEN_HALVED }

  The set of sparsification strategies supported by the ALP/Pregel interface.

Variables

- constexpr const SparsificationStrategy **out_sparsify** = NONE

  What sparsification strategy should be applied to the outgoing messages.

10.73.1 Detailed Description

This file defines a vertex-centric programming API called ALP/Pregel, which automatically translates to standard ALP/GraphBLAS primitives.

Author

A. N. Yzelman

Date

2022

10.74 pregel.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10 *
11 * Unless required by applicable law or agreed to in writing, software
12 * distributed under the License is distributed on an "AS IS" BASIS,
13 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 * See the License for the specific language governing permissions and
15 * limitations under the License.
16 */
17
142 #ifndef _H_GRB_INTERFACES_PREGEL
143 #define _H_GRB_INTERFACES_PREGEL
144
145 #include <graphblas.hpp>
146 #include <graphblas/utils/parser.hpp>
147
148 #include <stdexcept> // std::runtime_error
149
150
151 namespace grb {
152
153     namespace interfaces {
154
159         namespace config {
160
167             enum SparsificationStrategy {
168
174                 NONE = 0,
175
189                 ALWAYS,
190
209                 WHEN_REDUCED,
210
227                 WHEN_HALVED
228
229             };
230
242             constexpr const SparsificationStrategy out_sparsify = NONE;
243
244         } // end namespace grb::interfaces::config
245
266         struct PregelState {
267
282             bool &active;
283
291             bool &voteToHalt;
292
296             const size_t &num_vertices;
297
301             const size_t &num_edges;
302
306             const size_t &outdegree;
307
311             const size_t &indegree;
312
316             const size_t &round;
317
324             const size_t &vertexID;
325
326         };
327
337         template<
338             typename MatrixEntryType
339         >
340         class Pregel {
341
342             private:
343
345                 const size_t n;
346
348                 size_t nz;
349
351                 grb::Matrix< MatrixEntryType > graph;
352
354                 grb::Vector< bool > activeVertices;
```

```
355
357             grb::Vector< bool > haltVotes;
358
360             grb::Vector< bool > buffer;
361
363             grb::Vector< size_t > outdegrees;
364
366             grb::Vector< size_t > indegrees;
367
369             grb::Vector< size_t > IDs;
370
380             void initialize() {
381                 grb::Semiring<
382                     grb::operators::add< size_t >,
383                     grb::operators::right_assign_if< bool, size_t, size_t >,
384                     grb::identities::zero,
385                     grb::identities::logical_true
386                 > ring;
387                 grb::Vector< size_t > ones( n );
388                 if( grb::set( ones, 1 ) != SUCCESS ) {
389                     throw std::runtime_error( "Could not set vector ones" );
390                 }
391                 if( grb::set( outdegrees, 0 ) != SUCCESS ) {
392                     throw std::runtime_error( "Could not initialise outdegrees" );
393                 }
394                 if( grb::mxv< grb::descriptors::dense >(
395                         outdegrees, graph, ones, ring
396                     ) != SUCCESS
397                 ) {
398                     throw std::runtime_error( "Could not compute outdegrees" );
399                 }
400                 if( grb::set( indegrees, 0 ) != SUCCESS ) {
401                     throw std::runtime_error( "Could not initialise indegrees" );
402                 }
403                 if( grb::mxv<
404                     grb::descriptors::dense | grb::descriptors::transpose_matrix
405                 >(
406                     indegrees, graph, ones, ring
407                 ) != SUCCESS ) {
408                     throw std::runtime_error( "Could not compute indegrees" );
409                 }
410                 if( grb::set< grb::descriptors::use_index >(
411                         IDs, 0
412                     ) != SUCCESS
413                 ) {
414                     throw std::runtime_error( "Could not compute vertex IDs" );
415                 }
416             }
417
418
419         protected:
420
427             template< typename IType >
428             Pregel(
429                 const size_t _n,
430                 IType _start, const IType _end,
431                 const grb::IOMode _mode
432             ) :
433                 n( _n ),
434                 graph( _n, _n ),
435                 activeVertices( _n ),
436                 haltVotes( _n ),
437                 buffer( _n ),
438                 outdegrees( _n ),
439                 indegrees( _n ),
440                 IDs( _n )
441             {
442                 if( grb::ncols( graph ) != grb::nrows( graph ) ) {
443                     throw std::runtime_error( "Input graph is bipartite" );
444                 }
445                 if( grb::buildMatrixUnique(
446                     graph, _start, _end, _mode
447                 ) != SUCCESS ) {
448                     throw std::runtime_error( "Could not build graph" );
449                 }
450                 nz = grb::nnz( graph );
451                 initialize();
452             }
453
454
455         public:
456
500             template< typename IType >
501             Pregel(
502                 const size_t _m, const size_t _n,
503                 IType _start, const IType _end,
504                 const grb::IOMode _mode
```

```
505                 ) : Pregel( std::max( _m, _n ), _start, _end, _mode ) {}
506
641                 template<
642                     class Op,
643                     template< typename > class Id,
644                     class Program,
645                     typename IOType,
646                     typename GlobalProgramData,
647                     typename IncomingMessageType,
648                     typename OutgoingMessageType
649                 >
650                 grb::RC execute(
651                     const Program program,
652                     grb::Vector< IOType > &vertex_state,
653                     const GlobalProgramData &data,
654                     grb::Vector< IncomingMessageType > &in,
655                     grb::Vector< OutgoingMessageType > &out,
656                     size_t &rounds,
657                     grb::Vector< OutgoingMessageType > &out_buffer =
658                         grb::Vector< OutgoingMessageType >(0),
659                     const size_t max_rounds = 0
660                 ) {
661                     static_assert( grb::is_operator< Op >::value &&
662                             grb::is_associative< Op >::value,
663                         "The combiner must be an associate operator"
664                     );
665                     static_assert( std::is_same< typename Op::D1, IncomingMessageType >::value,
666                         "The combiner left-hand input domain should match the incoming message "
667                         "type." );
668                     static_assert( std::is_same< typename Op::D1, IncomingMessageType >::value,
669                         "The combiner right-hand input domain should match the incoming message "
670                         "type." );
671                     static_assert( std::is_same< typename Op::D1, IncomingMessageType >::value,
672                         "The combiner output domain should match the incoming message type." );
673
674                     // set default output
675                     rounds = 0;
676
677                     // sanity checks
678                     if( grb::size(vertex_state) != n ) {
679                         return MISMATCH;
680                     }
681                     if( grb::size(in) != n ) {
682                         return MISMATCH;
683                     }
684                     if( grb::size(out) != n ) {
685                         return MISMATCH;
686                     }
687                     if( grb::capacity(vertex_state) != n ) {
688                         return ILLEGAL;
689                     }
690                     if( grb::capacity(in) != n ) {
691                         return ILLEGAL;
692                     }
693                     if( grb::capacity(out) != n ) {
694                         return ILLEGAL;
695                     }
696                     if( config::out_sparsify && grb::capacity(out_buffer) != n ) {
697                         return ILLEGAL;
698                     }
699                     if( grb::nnz(vertex_state) != n ) {
700                         return ILLEGAL;
701                     }
702
703                     // define some monoids and semirings
704                     grb::Monoid<
705                         grb::operators::logical_or< bool >,
706                         grb::identities::logical_false
707                     > orMonoid;
708
709                     grb::Monoid<
710                         grb::operators::logical_and< bool >,
711                         grb::identities::logical_true
712                     > andMonoid;
713
714                     grb::Semiring<
715                         Op,
716                         grb::operators::left_assign_if<
717                             IncomingMessageType, bool, IncomingMessageType
718                         >,
719                         Id,
720                         grb::identities::logical_true
721                     > ring;
722
723                     // set initial round ID
724                     size_t step = 0;
725
```

```
726                     // activate all vertices
727                     grb::RC ret = grb::set( activeVertices, true );
728
729                     // initialise halt votes to all-false
730                     if( ret == SUCCESS ) {
731                         ret = grb::set( haltVotes, false );
732                     }
733
734                     // set default incoming message
735                     if( ret == SUCCESS && grb::nnz(in) < n ) {
736 #ifdef _DEBUG
737                         if( grb::nnz(in) > 0 ) {
738                             std::cerr « "Overwriting initial incoming messages since it was not a "
739                                 « "dense vector\n";
740                         }
741 #endif
742                         ret = grb::set( in, Id< IncomingMessageType >::value() );
743                     }
744
745                     // reset outgoing buffer
746                     size_t out_nnz = n;
747                     if( ret == SUCCESS ) {
748                         ret = grb::set( out, Id< OutgoingMessageType >::value() );
749                     }
750
751                     // return if initialisation failed
752                     if( ret != SUCCESS ) {
753                         assert( ret == FAILED );
754                         std::cerr « "Error: initialisation failed, but if workspace holds full "
755                             « "capacity, initialisation should never fail. Please submit a bug "
756                             « "report.\n";
757                         return PANIC;
758                     }
759
760                     // while there are active vertices, execute
761                     while( ret == SUCCESS ) {
762
763                         assert( max_rounds == 0 || step < max_rounds );
764                         // run one step of the program
765                         ret = grb::eWiseLambda(
766                             [
767                                 this,
768                                 &vertex_state,
769                                 &in,
770                                 &out,
771                                 &program,
772                                 step,
773                                 &data
774                             ]( const size_t i ) {
775                                 // create Pregel struct
776                                 PregelState pregel = {
777                                     activeVertices[ i ],
778                                     haltVotes[ i ],
779                                     n,
780                                     nz,
781                                     outdegrees[ i ],
782                                     indegrees[ i ],
783                                     step,
784                                     IDs[ i ]
785                                 };
786                                 // only execute program on active vertices
787                                 assert( activeVertices[ i ] );
788 #ifdef _DEBUG
789                                 std::cout « "Vertex " « i « " remains active in step " « step
790                                     « "\n";
791 #endif
792                                 program(
793                                     vertex_state[ i ],
794                                     in[ i ],
795                                     out[ i ],
796                                     data,
797                                     pregel
798                                 );
799 #ifdef _DEBUG
800                                 std::cout « "Vertex " « i « " sends out message " « out[ i ]
801                                     « "\n";
802 #endif
803                             }, activeVertices, vertex_state, in, out, outdegrees, haltVotes, indegrees,
       IDs
804                         );
805
806                         // increment counter
807                         (void) ++step;
808
809                         // check if everyone voted to halt
810                         if( ret == SUCCESS ) {
811                             bool halt = true;
```

```
812                             ret = grb::foldl< grb::descriptors::structural >(
813                                 halt, haltVotes, activeVertices, andMonoid
814                             );
815                             assert( ret == SUCCESS );
816                             if( ret == SUCCESS && halt ) {
817 #ifdef _DEBUG
818                                 std::cout « "\t All active vertices voted to halt; "
819                                     « "terminating Pregel program.\n";
820 #endif
821                                 break;
822                             }
823                         }
824 
825                         // update active vertices
826                         if( ret == SUCCESS ) {
827 #ifdef _DEBUG
828                             std::cout « "\t Number of active vertices was "
829                                 « grb::nnz( activeVertices ) « ", and ";
830 #endif
831                             ret = grb::clear( buffer );
832                             ret = ret ? ret : grb::set( buffer, activeVertices, true );
833                             std::swap( buffer, activeVertices );
834 #ifdef _DEBUG
835                             std::cout « " has now become " « grb::nnz( activeVertices ) « "\n";
836 #endif
837                         }
838 
839                         // check if there is a next round
840                         const size_t curActive = grb::nnz( activeVertices );
841                         if( ret == SUCCESS && curActive == 0 ) {
842 #ifdef _DEBUG
843                             std::cout « "\t All vertices are inactive; "
844                                 « "terminating Pregel program.\n";
845 #endif
846                             break;
847                         }
848 
849                         // check if we exceed the maximum number of rounds
850                         if( max_rounds > 0 && step > max_rounds ) {
851 #ifdef _DEBUG
852                             std::cout « "\t Maximum number of Pregel rounds met "
853                                 « "without the program returning a valid termination condition. "
854                                 « "Exiting prematurely with a FAILED error code.\n";
855 #endif
856                             ret = FAILED;
857                             break;
858                         }
859 
860 #ifdef _DEBUG
861                         std::cout « "\t Starting message exchange\n";
862 #endif
863 
864                         // reset halt votes
865                         if( ret == SUCCESS ) {
866                             ret = grb::clear( haltVotes );
867                             ret = ret ? ret : grb::set< grb::descriptors::structural >(
868                                 haltVotes, activeVertices, false
869                             );
870                         }
871 
872                         // reset incoming buffer
873                         if( ret == SUCCESS ) {
874                             ret = grb::clear( in );
875                             ret = ret ? ret : grb::set< grb::descriptors::structural >(
876                                 in, activeVertices, Id< IncomingMessageType >::value()
877                             );
878                         }
879 
880                         // execute communication
881                         if( ret == SUCCESS ) {
882                             ret = grb::vxm< grb::descriptors::structural >(
883                                 in, activeVertices, out, graph, ring
884                             );
885                         }
886 
887                         // sparsify and reset outgoing buffer
888                         if( config::out_sparsify && ret == SUCCESS ) {
889                             if( config::out_sparsify == config::ALWAYS ||
890                                 (config::out_sparsify == config::WHEN_REDUCED && out_nnz > curActive) ||
891                                 (config::out_sparsify == config::WHEN_HALVED && curActive <= out_nnz/2)
892                             ) {
893                                 ret = grb::clear( out_buffer );
894                                 ret = ret ? ret : grb::set< grb::descriptors::structural >(
895                                         out_buffer, activeVertices, Id< OutgoingMessageType >::value()
896                                     );
897                                 std::swap( out, out_buffer );
898                                 out_nnz = curActive;
```

```
899                             }
900                         }
901
902 #ifdef _DEBUG
903                         std::cout « "\t Resetting outgoing message fields and "
904                             « "starting next compute round\n";
905 #endif
906
907                     }
908
909 #ifdef _DEBUG
910                     if( grb::spmd<>::pid() == 0 ) {
911                         std::cout « "Info: Pregel exits after " « step
912                             « " rounds with error code " « ret
913                             « " ( " « grb::toString(ret) « " )\n";
914                     }
915 #endif
916
917                     // done
918                     rounds = step;
919                     return ret;
920                 }
921
928                 size_t num_vertices() const noexcept { return n; }
929
936                 size_t num_edges() const noexcept { return nz; }
937
949                 const grb::Matrix< MatrixEntryType > & get_matrix() const noexcept {
950                     return graph;
951                 }
952
953         };
954
955     } // end namespace "grb::interfaces"
956
957 } // end namespace "grb"
958
959 #endif // end "_H_GRB_INTERFACES_PREGEL"
960
```

10.75 iomode.hpp File Reference

Defines the various I/O modes a user could employ with ALP data ingestion or extraction.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Enumerations

- enum IOMode { SEQUENTIAL = 0 , PARALLEL }

  The GraphBLAS input and output functionalities can either be used in a sequential or parallel fashion.

10.75.1 Detailed Description

Defines the various I/O modes a user could employ with ALP data ingestion or extraction.

Author

A. N. Yzelman

Date

21st of February, 2017

10.76 iomode.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
28 #ifndef _H_GRB_IOMODE
29 #define _H_GRB_IOMODE
30
31
32 namespace grb {
33
67     enum IOMode {
68
75         SEQUENTIAL = 0,
76
92         PARALLEL
93     };
94
95 } // namespace grb
96
97 #endif // end "_H_GRB_IOMODE"
98
```

10.77 monoid.hpp File Reference

Provides an ALP monoid.

Classes

- class Monoid< _OP, _ID >

  A generalised monoid.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.77.1 Detailed Description

Provides an ALP monoid.

Author

A. N. Yzelman

Date

15 March, 2016

10.78 monoid.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_MONOID
28 #define _H_GRB_MONOID
29
30 #ifdef _DEBUG
31 #include <cstdio>
32 #endif
33
34 #include <cstddef> //size_t
35 #include <cstdlib> //posix_memalign, rand
36 #include <type_traits>
37
38 #include <assert.h>
39
40 #include <graphblas/identities.hpp>
41 #include <graphblas/ops.hpp>
42 #include <graphblas/type_traits.hpp>
43
44
45 namespace grb {
46
53     template< class _OP, template< typename > class _ID >
54     class Monoid {
55
56         static_assert( grb::is_operator< _OP >::value, "First template argument to Monoid must be a
    GraphBLAS operator" );
57
58         static_assert( grb::is_associative< _OP >::value,
59             "Cannot form a monoid using the given operator since it is not "
60             "associative" );
61
62         static_assert( std::is_same< typename _OP::D1, typename _OP::D3 >::value || std::is_same<
    typename _OP::D2, typename _OP::D3 >::value,
63             "Cannot form a monoid when the output domain does not match at least "
64             "one of its input domains" );
65
66     public:
67
69         typedef typename _OP::D1 D1;
70
72         typedef typename _OP::D2 D2;
73
75         typedef typename _OP::D3 D3;
76
78         typedef _OP Operator;
79
81         template< typename IdentityType >
82         using Identity = _ID< IdentityType >;
83
84
85     private:
86
92         Operator op;
93
94     public:
99         Monoid() : op() {}
100
110          template< typename D >
111          constexpr D getIdentity() const {
112              return Identity< D >::value();
113          }
114
120          Operator getOperator() const {
121              return op;
122          }
123      };
```

```
124
125     // type traits
126     template< class _OP, template< typename > class _ID >
127     struct is_monoid< Monoid< _OP, _ID > > {
129         static const constexpr bool value = true;
130     };
131
132     template< class OP, template< typename > class ID >
133     struct has_immutable_nonzeroes< Monoid< OP, ID > > {
134         static const constexpr bool value = grb::is_monoid< Monoid< OP, ID > >::value &&
135             std::is_same< OP, typename grb::operators::logical_or< typename OP::D1, typename OP::D2,
      typename OP::D3 > >::value;
136     };
137
138 } // namespace grb
139
140 #endif
141
```

10.79 ops.hpp File Reference

Provides a set of standard binary operators.

Classes

- class abs_diff< D1, D2, D3, implementation >

  This operator returns the absolute difference between two numbers.
- class add< D1, D2, D3, implementation >

  This operator takes the sum of the two input parameters and writes it to the output variable.
- class any_or< D1, D2, D3, implementation >

  This operator is a generalisation of the logical or.
- class argmax< IType, VType >

  The argmax operator on key-value pairs.
- class argmin< IType, VType >

  The argmin operator on key-value pairs.
- class divide< D1, D2, D3, implementation >

  Numerical division of two numbers.
- class divide_reverse< D1, D2, D3, implementation >

  Reversed division of two numbers.
- class equal< D1, D2, D3, implementation >

  Operator which returns `true` if its inputs compare equal, and `false` otherwise.
- class equal_first< D1, D2, D3, implementation >

  Compares `std::pair` inputs taking the first entry in every pair as the comparison key, and returns `true` or `false` accordingly.
- class geq< D1, D2, D3, implementation >

  This operation returns whether the left operand compares greater-than or equal to the right operand.
- class greater_than< D1, D2, D3, implementation >

  This operation returns whether the left operand compares greater-than the right operand.
- class left_assign< D1, D2, D3, implementation >

  This operator discards all right-hand side input and simply copies the left-hand side input to the output variable.
- class left_assign_if< D1, D2, D3, implementation >

  This operator assigns the left-hand input if the right-hand input evaluates `true`.
- class leq< D1, D2, D3, implementation >

  This operation returns whether the left operand compares less-than or equal to the right operand.
- class less_than< D1, D2, D3, implementation >

This operation returns whether the left operand compares less-than the right operand.

- class logical_and< D1, D2, D3, implementation >

  The logical and.
- class logical_or< D1, D2, D3, implementation >

  The logical or.
- class max< D1, D2, D3, implementation >

  This operator takes the maximum of the two input parameters and writes the result to the output variable.
- class min< D1, D2, D3, implementation >

  This operator takes the minimum of the two input parameters and writes the result to the output variable.
- class mul< D1, D2, D3, implementation >

  This operator multiplies the two input parameters and writes the result to the output variable.
- class not_equal< D1, D2, D3, implementation >

  Operator that returns `false` whenever its inputs compare equal, and `true` otherwise.
- class relu< D1, D2, D3, implementation >

  This operation is equivalent to grb::operators::min.
- class right_assign< D1, D2, D3, implementation >

  This operator discards all left-hand side input and simply copies the right-hand side input to the output variable.
- class right_assign_if< D1, D2, D3, implementation >

  This operator assigns the right-hand input if the left-hand input evaluates `true`.
- class square_diff< D1, D2, D3, implementation >

  This operation returns the squared difference between two numbers.
- class subtract< D1, D2, D3, implementation >

  Numerical substraction of two numbers.
- class zip< IN1, IN2, implementation >

  The zip operator that operators on keys as a left-hand input and values as a right hand input, producing a key-value `std::pair`.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.
- namespace grb::operators

  This namespace holds various standard operators such as grb::operators::add and grb::operators::mul.

Macros

- #define _DEBUG_NO_IOSTREAM_PAIR_CONVERTER

  Macro that disables the definition of an operator<< overload for instances of std::pair.

10.79.1 Detailed Description

Provides a set of standard binary operators.

Author

A. N. Yzelman

Date

8th of August, 2016

10.79.2 Macro Definition Documentation

10.79.2.1 _DEBUG_NO_IOSTREAM_PAIR_CONVERTER

```
#define _DEBUG_NO_IOSTREAM_PAIR_CONVERTER
```

Macro that disables the definition of an operator<< overload for instances of std::pair.

This overload is only active when the _DEBUG macro is defined, but may clash with user-defined overloads.

10.80 ops.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_OPERATORS
28 #define _H_GRB_OPERATORS
29
30 #include "internalops.hpp"
31 #include "type_traits.hpp"
32
33
34 namespace grb {
35
40     namespace operators {
41
54         template<
55             typename D1, typename D2 = D1, typename D3 = D2,
56             enum Backend implementation = config::default_backend
57         >
58         class left_assign :
59             public internal::Operator<
60             internal::left_assign< D1, D2, D3, implementation >
61         > {
62
63             public:
64
65                 template< typename A, typename B, typename C, enum Backend D >
66                 using GenericOperator = left_assign< A, B, C, D >;
67
68                 left_assign() {}
69
70         };
71
81         template<
82             typename D1, typename D2 = D1, typename D3 = D2,
83             enum Backend implementation = config::default_backend
84         >
85         class left_assign_if :
86              public internal::Operator<
87                     internal::left_assign_if< D1, D2, D3, implementation >
88         > {
89
90             public:
91
92                 template< typename A, typename B, typename C, enum Backend D >
```

```
93                 using GenericOperator = left_assign_if< A, B, C, D >;
94
95                 left_assign_if() {}
96
97         };
98
111        template<
112            typename D1, typename D2 = D1, typename D3 = D2,
113            enum Backend implementation = config::default_backend
114        >
115        class right_assign : public internal::Operator<
116                internal::right_assign< D1, D2, D3, implementation >
117        > {
118
119            public:
120
121                template< typename A, typename B, typename C, enum Backend D >
122                using GenericOperator = right_assign< A, B, C, D >;
123
124                right_assign() {}
125
126        };
127
137        template<
138            typename D1, typename D2 = D1, typename D3 = D2,
139            enum Backend implementation = config::default_backend
140        >
141        class right_assign_if : public internal::Operator<
142                internal::right_assign_if< D1, D2, D3, implementation >
143        > {
144
145            public:
146
147                template< typename A, typename B, typename C, enum Backend D >
148                using GenericOperator = right_assign_if< A, B, C, D >;
149
150                right_assign_if() {}
151
152        };
153
170        // [Operator Wrapping]
171        template<
172            typename D1, typename D2 = D1, typename D3 = D2,
173            enum Backend implementation = config::default_backend
174        >
175        class add : public internal::Operator<
176            internal::add< D1, D2, D3, implementation >
177        > {
178
179            public:
180
181                template< typename A, typename B, typename C, enum Backend D >
182                using GenericOperator = add< A, B, C, D >;
183
184                add() {}
185        };
186        // [Operator Wrapping]
187
204        template<
205            typename D1, typename D2 = D1, typename D3 = D2,
206            enum Backend implementation = config::default_backend
207        >
208        class mul : public internal::Operator<
209            internal::mul< D1, D2, D3, implementation >
210        > {
211
212            public:
213
214                template< typename A, typename B, typename C, enum Backend D >
215                using GenericOperator = mul< A, B, C, D >;
216
217                mul() {}
218        };
219
237        template<
238            typename D1, typename D2 = D1, typename D3 = D2,
239            enum Backend implementation = config::default_backend
240        >
241        class max : public internal::Operator<
242            internal::max< D1, D2, D3, implementation >
243        > {
244
245            public:
246
247                template< typename A, typename B, typename C, enum Backend D >
248                using GenericOperator = max< A, B, C, D >;
249
```

```
250                 max() {}
251         };
252
270         template<
271             typename D1, typename D2 = D1, typename D3 = D2,
272             enum Backend implementation = config::default_backend
273         >
274         class min : public internal::Operator<
275                 internal::min< D1, D2, D3, implementation >
276         > {
277
278             public:
279
280                 template< typename A, typename B, typename C, enum Backend D >
281                 using GenericOperator = min< A, B, C, D >;
282
283                 min() {}
284         };
285
297         template<
298             typename D1, typename D2 = D1, typename D3 = D2,
299             enum Backend implementation = config::default_backend
300         >
301         class subtract : public internal::Operator<
302             internal::substract< D1, D2, D3, implementation >
303         > {
304
305             public:
306
307                 template< typename A, typename B, typename C, enum Backend D >
308                 using GenericOperator = subtract< A, B, C, D >;
309
310                 subtract() {}
311         };
312
324         template<
325             typename D1, typename D2 = D1, typename D3 = D2,
326             enum Backend implementation = config::default_backend
327         >
328         class divide : public internal::Operator<
329                 internal::divide< D1, D2, D3, implementation >
330         > {
331
332             public:
333
334                 template< typename A, typename B, typename C, enum Backend D >
335                 using GenericOperator = divide< A, B, C, D >;
336
337                 divide() {}
338         };
339
349         template<
350             typename D1, typename D2 = D1, typename D3 = D2,
351             enum Backend implementation = config::default_backend
352         >
353         class divide_reverse : public internal::Operator<
354                 internal::divide_reverse< D1, D2, D3, implementation >
355         > {
356
357             public:
358
359                 template< typename A, typename B, typename C, enum Backend D >
360                 using GenericOperator = divide_reverse< A, B, C, D >;
361
362                 divide_reverse() {}
363         };
364
375         template<
376             typename D1, typename D2 = D1, typename D3 = D2,
377             enum Backend implementation = config::default_backend
378         >
379         class equal : public internal::Operator<
380                 internal::equal< D1, D2, D3, implementation >
381         > {
382
383             public:
384
385                 template< typename A, typename B, typename C, enum Backend D >
386                 using GenericOperator = equal< A, B, C, D >;
387
388                 equal() {}
389         };
390
401         template<
402             typename D1, typename D2 = D1, typename D3 = D2,
403             enum Backend implementation = config::default_backend
404         >
```

```
405         class not_equal : public internal::Operator<
406             internal::not_equal< D1, D2, D3, implementation >
407         > {
408 
409             public:
410 
411                 template< typename A, typename B, typename C, enum Backend D >
412                 using GenericOperator = not_equal< A, B, C, D >;
413 
414                 not_equal() {}
415         };
416 
430         template<
431             typename D1, typename D2 = D1, typename D3 = D2,
432             enum Backend implementation = config::default_backend
433         >
434         class any_or : public internal::Operator<
435             internal::any_or< D1, D2, D3, implementation >
436         > {
437 
438             public:
439 
440                 template< typename A, typename B, typename C, enum Backend D >
441                 using GenericOperator = any_or< A, B, C, D >;
442 
443                 any_or() {}
444         };
445 
458         template<
459             typename D1, typename D2 = D1, typename D3 = D2,
460             enum Backend implementation = config::default_backend
461         >
462         class logical_or : public internal::Operator<
463                 internal::logical_or< D1, D2, D3, implementation >
464         > {
465 
466             public:
467 
468                 template< typename A, typename B, typename C, enum Backend D >
469                 using GenericOperator = logical_or< A, B, C, D >;
470 
471                 logical_or() {}
472         };
473 
486         template<
487             typename D1, typename D2 = D1, typename D3 = D2,
488             enum Backend implementation = config::default_backend
489         >
490         class logical_and : public internal::Operator<
491                 internal::logical_and< D1, D2, D3, implementation >
492         > {
493 
494             public:
495 
496                 template< typename A, typename B, typename C, enum Backend D >
497                 using GenericOperator = logical_and< A, B, C, D >;
498 
499                 logical_and() {}
500         };
501 
510         template<
511             typename D1, typename D2 = D1, typename D3 = D2,
512             enum Backend implementation = config::default_backend
513         >
514         class relu : public internal::Operator<
515                 internal::relu< D1, D2, D3, implementation >
516         > {
517 
518             public:
519 
520                 template< typename A, typename B, typename C, enum Backend D >
521                 using GenericOperator = relu< A, B, C, D >;
522 
523                 relu() {}
524         };
525 
537         template<
538             typename D1, typename D2 = D1, typename D3 = D2,
539             enum Backend implementation = config::default_backend
540         >
541         class abs_diff : public internal::Operator<
542                 internal::abs_diff< D1, D2, D3, implementation >
543         > {
544 
545             public:
546 
547                 template< typename A, typename B, typename C, enum Backend D >
```

```
548                 using GenericOperator = abs_diff< A, B, C, D >;
549
550                 abs_diff() {}
551
552         };
553
572         template< typename IType, typename VType >
573         class argmin : public internal::Operator< internal::argmin< IType, VType > > {
574
575             public:
576
577                 argmin() {}
578
579         };
580
599         template< typename IType, typename VType >
600         class argmax : public internal::Operator< internal::argmax< IType, VType > > {
601
602             public:
603
604                 argmax() {}
605
606         };
607
619         template<
620             typename D1, typename D2, typename D3,
621             enum Backend implementation = config::default_backend
622         >
623         class square_diff : public internal::Operator<
624                 internal::square_diff< D1, D2, D3, implementation >
625         > {
626
627             public:
628
629                 template< typename A, typename B, typename C, enum Backend D >
630                 using GenericOperator = square_diff< A, B, C, D >;
631
632                 square_diff() {}
633
634         };
635
645         template<
646             typename IN1, typename IN2,
647             enum Backend implementation = config::default_backend
648         >
649         class zip : public internal::Operator<
650                 internal::zip< IN1, IN2, implementation >
651         > {
652
653             public:
654
655                 template< typename A, typename B, enum Backend D >
656                 using GenericOperator = zip< A, B, D >;
657
658                 zip() {}
659
660         };
661
674         template<
675             typename D1, typename D2 = D1, typename D3 = D2,
676             enum Backend implementation = config::default_backend
677         >
678         class equal_first : public internal::Operator<
679                 internal::equal_first< D1, D2, D3, implementation >
680         > {
681
682             public:
683
684                 template< typename A, typename B, typename C, enum Backend D >
685                 using GenericOperator = equal_first< A, B, C, D >;
686
687                 equal_first() {}
688
689         };
690
703         template<
704             typename D1, typename D2 = D1, typename D3 = D2,
705             enum Backend implementation = config::default_backend
706         >
707         class less_than : public internal::Operator<
708                 internal::lt< D1, D2, D3, implementation >
709         > {
710
711             public:
712
713                 template< typename A, typename B, typename C, enum Backend D >
714                 using GenericOperator = less_than< A, B, C, D >;
```

```
715
716                 less_than() {}
717
718         };
719
732         template<
733             typename D1, typename D2 = D1, typename D3 = D2,
734             enum Backend implementation = config::default_backend
735         >
736         class leq : public internal::Operator<
737                 internal::leq< D1, D2, D3, implementation >
738         > {
739
740             public:
741
742                 template< typename A, typename B, typename C, enum Backend D >
743                 using GenericOperator = leq< A, B, C, D >;
744
745                 leq() {}
746
747         };
748
761         template<
762             typename D1, typename D2 = D1, typename D3 = D2,
763             enum Backend implementation = config::default_backend
764         >
765         class greater_than: public internal::Operator<
766                 internal::gt< D1, D2, D3, implementation >
767         > {
768
769             public:
770
771                 template< typename A, typename B, typename C, enum Backend D >
772                 using GenericOperator = greater_than< A, B, C, D >;
773
774                 greater_than() {}
775
776         };
777
790         template<
791             typename D1, typename D2 = D1, typename D3 = D2,
792             enum Backend implementation = config::default_backend
793         >
794         class geq : public internal::Operator<
795                 internal::geq< D1, D2, D3, implementation >
796         > {
797
798             public:
799
800                 template< typename A, typename B, typename C, enum Backend D >
801                 using GenericOperator = geq< A, B, C, D >;
802
803                 geq() {}
804
805         };
806
807     } // namespace operators
808
809     template< typename D1, typename D2, typename D3, enum Backend implementation >
810     struct is_operator< operators::left_assign_if< D1, D2, D3, implementation > > {
811         static const constexpr bool value = true;
812     };
813
814     template< typename D1, typename D2, typename D3, enum Backend implementation >
815     struct is_operator< operators::right_assign_if< D1, D2, D3, implementation > > {
816         static const constexpr bool value = true;
817     };
818
819     template< typename D1, typename D2, typename D3, enum Backend implementation >
820     struct is_operator< operators::left_assign< D1, D2, D3, implementation > > {
821         static const constexpr bool value = true;
822     };
823
824     template< typename D1, typename D2, typename D3, enum Backend implementation >
825     struct is_operator< operators::right_assign< D1, D2, D3, implementation > > {
826         static const constexpr bool value = true;
827     };
828
829     // [Operator Type Traits]
830     template< typename D1, typename D2, typename D3, enum Backend implementation >
831     struct is_operator< operators::add< D1, D2, D3, implementation > > {
832         static const constexpr bool value = true;
833     };
834     // [Operator Type Traits]
835
836     template< typename D1, typename D2, typename D3, enum Backend implementation >
837     struct is_operator< operators::mul< D1, D2, D3, implementation > > {
```

```
838         static const constexpr bool value = true;
839     };
840
841     template< typename D1, typename D2, typename D3, enum Backend implementation >
842     struct is_operator< operators::max< D1, D2, D3, implementation > > {
843         static const constexpr bool value = true;
844     };
845
846     template< typename D1, typename D2, typename D3, enum Backend implementation >
847     struct is_operator< operators::min< D1, D2, D3, implementation > > {
848         static const constexpr bool value = true;
849     };
850
851     template< typename D1, typename D2, typename D3, enum Backend implementation >
852     struct is_operator< operators::subtract< D1, D2, D3, implementation > > {
853         static const constexpr bool value = true;
854     };
855
856     template< typename D1, typename D2, typename D3, enum Backend implementation >
857     struct is_operator< operators::divide< D1, D2, D3, implementation > > {
858         static const constexpr bool value = true;
859     };
860
861     template< typename D1, typename D2, typename D3, enum Backend implementation >
862     struct is_operator< operators::divide_reverse< D1, D2, D3, implementation > > {
863         static const constexpr bool value = true;
864     };
865
866     template< typename D1, typename D2, typename D3, enum Backend implementation >
867     struct is_operator< operators::equal< D1, D2, D3, implementation > > {
868         static const constexpr bool value = true;
869     };
870
871     template< typename D1, typename D2, typename D3, enum Backend implementation >
872     struct is_operator< operators::not_equal< D1, D2, D3, implementation > > {
873         static const constexpr bool value = true;
874     };
875
876     template< typename D1, typename D2, typename D3, enum Backend implementation >
877     struct is_operator< operators::any_or< D1, D2, D3, implementation > > {
878         static const constexpr bool value = true;
879     };
880
881     template< typename D1, typename D2, typename D3, enum Backend implementation >
882     struct is_operator< operators::logical_or< D1, D2, D3, implementation > > {
883         static const constexpr bool value = true;
884     };
885
886     template< typename D1, typename D2, typename D3, enum Backend implementation >
887     struct is_operator< operators::logical_and< D1, D2, D3, implementation > > {
888         static const constexpr bool value = true;
889     };
890
891     template< typename D1, typename D2, typename D3, enum Backend implementation >
892     struct is_operator< operators::abs_diff< D1, D2, D3, implementation > > {
893         static const constexpr bool value = true;
894     };
895
896     template< typename D1, typename D2, typename D3, enum Backend implementation >
897     struct is_operator< operators::relu< D1, D2, D3, implementation > > {
898         static const constexpr bool value = true;
899     };
900
901     template< typename IType, typename VType >
902     struct is_operator< operators::argmin< IType, VType > > {
903         static const constexpr bool value = true;
904     };
905
906     template< typename IType, typename VType >
907     struct is_operator< operators::argmax< IType, VType > > {
908         static const constexpr bool value = true;
909     };
910
911     template< typename D1, typename D2, typename D3, enum Backend implementation >
912     struct is_operator< operators::square_diff< D1, D2, D3, implementation > > {
913         static const constexpr bool value = true;
914     };
915
916     template< typename IN1, typename IN2, enum Backend implementation >
917     struct is_operator< operators::zip< IN1, IN2, implementation > > {
918         static const constexpr bool value = true;
919     };
920
921     template< typename D1, typename D2, typename D3, enum Backend implementation >
922     struct is_operator< operators::equal_first< D1, D2, D3, implementation > > {
923         static const constexpr bool value = true;
924     };
```

```
925
926     template< typename D1, typename D2, typename D3, enum Backend implementation >
927     struct is_operator< operators::less_than< D1, D2, D3, implementation > > {
928         static const constexpr bool value = true;
929     };
930
931     template< typename D1, typename D2, typename D3, enum Backend implementation >
932     struct is_operator< operators::leq< D1, D2, D3, implementation > > {
933         static const constexpr bool value = true;
934     };
935
936     template< typename D1, typename D2, typename D3, enum Backend implementation >
937     struct is_operator< operators::greater_than< D1, D2, D3, implementation > > {
938         static const constexpr bool value = true;
939     };
940
941     template< typename D1, typename D2, typename D3, enum Backend implementation >
942     struct is_operator< operators::geq< D1, D2, D3, implementation > > {
943         static const constexpr bool value = true;
944     };
945
946     template< typename D1, typename D2, typename D3 >
947     struct is_idempotent< operators::min< D1, D2, D3 >, void > {
948         static const constexpr bool value = true;
949     };
950
951     template< typename D1, typename D2, typename D3 >
952     struct is_idempotent< operators::max< D1, D2, D3 >, void > {
953         static const constexpr bool value = true;
954     };
955
956     template< typename D1, typename D2, typename D3 >
957     struct is_idempotent< operators::any_or< D1, D2, D3 >, void > {
958         static const constexpr bool value = true;
959     };
960
961     template< typename D1, typename D2, typename D3 >
962     struct is_idempotent< operators::logical_or< D1, D2, D3 >, void > {
963         static const constexpr bool value = true;
964     };
965
966     template< typename D1, typename D2, typename D3 >
967     struct is_idempotent< operators::logical_and< D1, D2, D3 >, void > {
968         static const constexpr bool value = true;
969     };
970
971     template< typename D1, typename D2, typename D3 >
972     struct is_idempotent< operators::relu< D1, D2, D3 >, void > {
973         static const constexpr bool value = true;
974     };
975
976     template< typename D1, typename D2, typename D3 >
977     struct is_idempotent< operators::left_assign_if< D1, D2, D3 >, void > {
978         static const constexpr bool value = true;
979     };
980
981     template< typename D1, typename D2, typename D3 >
982     struct is_idempotent< operators::right_assign_if< D1, D2, D3 >, void > {
983         static const constexpr bool value = true;
984     };
985
986     template< typename IType, typename VType >
987     struct is_idempotent< operators::argmin< IType, VType >, void > {
988         static const constexpr bool value = true;
989     };
990
991     template< typename IType, typename VType >
992     struct is_idempotent< operators::argmax< IType, VType >, void > {
993         static const constexpr bool value = true;
994     };
995
996     template< typename OP >
997     struct is_associative<
998         OP,
999         typename std::enable_if< is_operator< OP >::value, void >::type
1000     > {
1001         static constexpr const bool value = OP::is_associative();
1002     };
1003
1004     template< typename OP >
1005     struct is_commutative<
1006         OP,
1007         typename std::enable_if< is_operator< OP >::value, void >::type
1008     > {
1009         static constexpr const bool value = OP::is_commutative();
1010     };
1011
```

```
1012     // internal type traits follow
1013
1014     namespace internal {
1015
1016         template< typename D1, typename D2, typename D3, enum Backend implementation >
1017         struct maybe_noop< operators::left_assign_if< D1, D2, D3, implementation > > {
1018             static const constexpr bool value = true;
1019         };
1020
1021         template< typename D1, typename D2, typename D3, enum Backend implementation >
1022         struct maybe_noop< operators::right_assign_if< D1, D2, D3, implementation > > {
1023             static const constexpr bool value = true;
1024         };
1025
1026     } // end namespace grb::internal
1027
1028 } // end namespace grb
1029
1030 #ifdef __DOXYGEN__
1036  #define _DEBUG_NO_IOSTREAM_PAIR_CONVERTER
1037 #endif
1038
1039 #ifdef _DEBUG
1040  #ifndef _DEBUG_NO_IOSTREAM_PAIR_CONVERTER
1041 namespace std {
1042     template< typename U, typename V >
1043     std::ostream & operator«( std::ostream &out, const std::pair< U, V > &pair ) {
1044         out « "( " « pair.first « ", " « pair.second « " )";
1045         return out;
1046     }
1047 } // end namespace std
1048  #endif
1049 #endif
1050
1051 #endif // end "_H_GRB_OPERATORS"
1052
```

10.81 phase.hpp File Reference

Defines the various phases an ALP/GraphBLAS primitive may be executed with.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Enumerations

- enum Phase { RESIZE , TRY , EXECUTE }

  Primitives with sparse ALP/GraphBLAS output containers may run into the issue where an appropriate grb::capacity may not always be clear.

10.81.1 Detailed Description

Defines the various phases an ALP/GraphBLAS primitive may be executed with.

Author

A. N. Yzelman

10.82 phase.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
26 #ifndef _H_GRB_PHASE
27 #define _H_GRB_PHASE
28
29
30 namespace grb {
31
152     enum Phase {
153
187         RESIZE,
188
225         TRY,
226
257         EXECUTE
258
259     };
260
261 } // namespace grb
262
263 #endif // end "_H_GRB_PHASE"
264
```

10.83 rc.hpp File Reference

Defines the ALP error codes.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

Enumerations

- enum RC {
  SUCCESS = 0 , PANIC , OUTOFMEM , MISMATCH ,
  OVERLAP , OVERFLW , UNSUPPORTED , ILLEGAL ,
  FAILED }

  Return codes of ALP primitives.

Functions

- std::string toString (const RC code)

10.83.1 Detailed Description

Defines the ALP error codes.

Author

A. N. Yzelman

Date

9–11 August, 2016

10.84 rc.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_RC
28 #define _H_GRB_RC
29
30 #include <string>
31
32
33 namespace grb {
34
47     enum RC {
48
54         SUCCESS = 0,
55
68         PANIC,
69
78         OUTOFMEM,
79
90         MISMATCH,
91
106         OVERLAP,
107
119         OVERFLW,
120
129         UNSUPPORTED,
130
143         ILLEGAL,
144
154         FAILED
155
156     };
157
159     std::string toString( const RC code );
160
161 } // namespace grb
162
163 #endif
164
```

10.85 semiring.hpp File Reference

Provides an ALP semiring.

Classes

- class Semiring< _OP1, _OP2, _ID1, _ID2 >

  A generalised semiring.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.85.1 Detailed Description

Provides an ALP semiring.

Author

A. N. Yzelman

Date

15th of March, 2016

10.86 semiring.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_SEMIRING
28 #define _H_GRB_SEMIRING
29
30 #include <graphblas/identities.hpp>
31 #include <graphblas/monoid.hpp>
32 #include <graphblas/ops.hpp>
33
34
35 namespace grb {
36
185     template< class _OP1, class _OP2, template< typename > class _ID1, template< typename > class _ID2 >
186     class Semiring {
187
188         static_assert( std::is_same< typename _OP2::D3, typename _OP1::D1 >::value,
189             "The multiplicative output type must match the left-hand additive "
190             "input type" );
191
192         static_assert( std::is_same< typename _OP1::D2, typename _OP1::D3 >::value,
193             "The right-hand input type of the additive operator must match its "
194             "output type" );
195
196         static_assert( grb::is_associative< _OP1 >::value,
197             "Cannot construct a semiring using a non-associative additive "
```

```
198             "operator" );
199
200         static_assert( grb::is_associative< _OP2 >::value,
201             "Cannot construct a semiring using a non-associative multiplicative "
202             "operator" );
203
204         static_assert( grb::is_commutative< _OP1 >::value,
205             "Cannot construct a semiring using a non-commutative additive "
206             "operator" );
207
208     public:
210         typedef typename _OP2::D1 D1;
211
213         typedef typename _OP2::D2 D2;
214
219         typedef typename _OP2::D3 D3;
220
225         typedef typename _OP1::D2 D4;
226
228         typedef _OP1 AdditiveOperator;
229
231         typedef _OP2 MultiplicativeOperator;
232
234         typedef Monoid< _OP1, _ID1 > AdditiveMonoid;
235
237         typedef Monoid< _OP2, _ID2 > MultiplicativeMonoid;
238
240         template< typename ZeroType >
241         using Zero = _ID1< ZeroType >;
242
244         template< typename OneType >
245         using One = _ID2< OneType >;
246
247     private:
248         static constexpr size_t D1_bsz = grb::config::SIMD_BLOCKSIZE< D1 >::value();
249         static constexpr size_t D2_bsz = grb::config::SIMD_BLOCKSIZE< D2 >::value();
250         static constexpr size_t D3_bsz = grb::config::SIMD_BLOCKSIZE< D3 >::value();
251         static constexpr size_t D4_bsz = grb::config::SIMD_BLOCKSIZE< D4 >::value();
252         static constexpr size_t mul_input_bsz = D1_bsz < D2_bsz ? D1_bsz : D2_bsz;
253
255         AdditiveMonoid additiveMonoid;
256
258         MultiplicativeMonoid multiplicativeMonoid;
259
260     public:
262         static constexpr size_t blocksize_add = D3_bsz < D4_bsz ? D3_bsz : D4_bsz;
263
265         static constexpr size_t blocksize_mul = mul_input_bsz < D3_bsz ? mul_input_bsz : D3_bsz;
266
268         static constexpr size_t blocksize = blocksize_mul < blocksize_add ? blocksize_mul :
    blocksize_add;
269
282         template< typename D >
283         constexpr D getZero() const {
284             return additiveMonoid.template getIdentity< D >();
285         }
286
298         template< typename D >
299         constexpr D getOne() const {
300             return multiplicativeMonoid.template getIdentity< D >();
301         }
302
308         AdditiveMonoid getAdditiveMonoid() const {
309             return additiveMonoid;
310         }
311
317         MultiplicativeMonoid getMultiplicativeMonoid() const {
318             return multiplicativeMonoid;
319         }
320
326         AdditiveOperator getAdditiveOperator() const {
327             return additiveMonoid.getOperator();
328         }
329
335         MultiplicativeOperator getMultiplicativeOperator() const {
336             return multiplicativeMonoid.getOperator();
337         }
338     };
339
340     // overload for GraphBLAS type traits.
341     template< class _OP1, class _OP2, template< typename > class _ID1, template< typename > class _ID2 >
342     struct is_semiring< Semiring< _OP1, _OP2, _ID1, _ID2 > > {
344         static const constexpr bool value = true;
345     };
346
347     template< class _OP1, class _OP2, template< typename > class _ID1, template< typename > class _ID2 >
348     struct has_immutable_nonzeroes< Semiring< _OP1, _OP2, _ID1, _ID2 > > {
```

```
349         static const constexpr bool value = grb::is_semiring< Semiring< _OP1, _OP2, _ID1, _ID2 >
      >::value &&
350             std::is_same< _OP1, typename grb::operators::logical_or< typename _OP1::D1, typename
      _OP1::D2, typename _OP1::D3 > >::value;
351       };
352
353 } // namespace grb
354
355 #endif
356
```

10.87 type_traits.hpp File Reference

Specifies the ALP algebraic type traits.

Classes

- struct has_immutable_nonzeroes< T >

  Used to inspect whether a given semiring has immutable nonzeroes under addition.
- struct is_associative< T, typename >

  Used to inspect whether a given operator or monoid is associative.
- struct is_commutative< T, typename >

  Used to inspect whether a given operator or monoid is commutative.
- struct is_container< T >

  Used to inspect whether a given type is an ALP/GraphBLAS container.
- struct is_idempotent< T, typename >

  Used to inspect whether a given operator or monoid is idempotent.
- struct is_monoid< T >

  Used to inspect whether a given type is an ALP monoid.
- struct is_object< T >

  Used to inspect whether a given type is an ALP/GraphBLAS object.
- struct is_operator< T >

  Used to inspect whether a given type is an ALP operator.
- struct is_semiring< T >

  Used to inspect whether a given type is an ALP semiring.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.87.1 Detailed Description

Specifies the ALP algebraic type traits.

Author

A. N. Yzelman

Date

25th of March, 2019

10.88 type_traits.hpp

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
27 #ifndef _H_GRB_TYPE_TRAITS
28 #define _H_GRB_TYPE_TRAITS
29
30 #include <type_traits>
31
32
33 namespace grb {
34
46     template< typename T >
47     struct is_container {
48
54         static const constexpr bool value = false;
55
56     };
57
65     template< typename T >
66     struct is_semiring {
67
73         static const constexpr bool value = false;
74
75     };
76
84     template< typename T >
85     struct is_monoid {
86
92         static const constexpr bool value = false;
93
94     };
95
103     template< typename T >
104     struct is_operator {
105
111         static const constexpr bool value = false;
112     };
113
129     template< typename T >
130     struct is_object {
131
135         static const constexpr bool value = is_container< T >::value ||
136             is_semiring< T >::value ||
137             is_monoid< T >::value ||
138             is_operator< T >::value;
139     };
140
160     template< typename T, typename = void >
161     struct is_idempotent {
162
163         static_assert( is_operator< T >::value || is_monoid< T >::value,
164             "Template argument to grb::is_idempotent must be an operator or a monoid!" );
165
167         static const constexpr bool value = false;
168
169     };
170
178     template< typename Monoid >
179     struct is_idempotent<
180         Monoid,
181         typename std::enable_if< is_monoid< Monoid >::value, void >::type
182     > {
183         static const constexpr bool value =
184             is_idempotent< typename Monoid::Operator >::value;
185     };
186
201     template< typename T, typename = void >
202     struct is_associative {
```

```
203
204         static_assert( is_operator< T >::value || is_monoid< T >::value,
205             "Template argument should be an ALP binary operator or monoid." );
206
208         static const constexpr bool value = false;
209
210     };
211
219     template< typename Monoid >
220     struct is_associative<
221         Monoid,
222         typename std::enable_if< is_monoid< Monoid >::value, void >::type
223     > {
224         static_assert( is_associative< typename Monoid::Operator >::value,
225             "Malformed ALP monoid encountered" );
226         static const constexpr bool value = true;
227     };
228
239     template< typename T, typename = void >
240     struct is_commutative {
241
242         static_assert( is_operator< T >::value || is_monoid< T >::value,
243             "Template argument should be an ALP binary operator or monoid." );
244
246         static const constexpr bool value = false;
247
248     };
249
257     template< typename Monoid >
258     struct is_commutative<
259         Monoid,
260         typename std::enable_if< is_monoid< Monoid >::value, void >::type
261     > {
263         static const constexpr bool value =
264             is_commutative< typename Monoid::Operator >::value;
265     };
266
278     template< typename T >
279     struct has_immutable_nonzeroes {
280
281         static_assert( is_semiring< T >::value,
282             "Template argument to grb::has_immutable_nonzeroes must be a "
283             "semiring!" );
284
286         static const constexpr bool value = false;
287
288     };
289
290     namespace internal {
291
304         template< typename OP >
305         struct maybe_noop {
306             static_assert( is_operator< OP >::value,
307                 "Argument to internal::maybe_noop must be an operator."
308             );
309             static const constexpr bool value = false;
310         };
311
312     } // end namespace grb::internal
313
314 } // namespace grb
315
316 #endif // end _H_GRB_TYPE_TRAITS
317
```

10.89 blas_sparse.h File Reference

This is the ALP implementation of a subset of the NIST Sparse BLAS standard.

Typedefs

- typedef void * blas_sparse_matrix

  A sparse matrix.

Enumerations

- enum blas_order_type

  The supported dense storages.
- enum blas_trans_type

  The possible transposition types.

Functions

- blas_sparse_matrix BLAS_duscr_begin (const int m, const int n)

  Creates a handle to a new / empty sparse matrix.
- int BLAS_duscr_end (blas_sparse_matrix A)

  Signals that the matrix A can now be finalised – all contents have been added.
- int BLAS_duscr_insert_col (blas_sparse_matrix A, const int j, const int nnz, const double ∗vals, const int ∗rows)

  Inserts a column into A.
- int BLAS_duscr_insert_entries (blas_sparse_matrix A, const int nnz, const double ∗vals, const int ∗rows, const int ∗cols)

  Inserts a block of entries into A.
- int BLAS_duscr_insert_entry (blas_sparse_matrix A, const double val, const int row, const int col)

  Inserts a single nonzero entry into A.
- int BLAS_duscr_insert_row (blas_sparse_matrix A, const int i, const int nnz, const double ∗vals, const int ∗cols)

  Inserts a row into A.
- int BLAS_dusmm (const enum blas_order_type order, const enum blas_trans_type transa, const int nrhs, const double alpha, const blas_sparse_matrix A, const double ∗B, const int ldb, const double ∗C, const int ldc)

  Sparse matrix–dense matrix multiplication.
- int BLAS_dusmv (const enum blas_trans_type transa, const double alpha, const blas_sparse_matrix A, const double ∗const x, int incx, double ∗const y, const int incy)

  Sparse matrix–dense vector multiplication.
- int BLAS_usds (blas_sparse_matrix A)

  Frees a given matrix.
- int EXTBLAS_dusm_clear (blas_sparse_matrix A)

  Removes all entries from a finalised sparse matrix.
- int EXTBLAS_dusm_close (const blas_sparse_matrix A)

  Closes a sparse matrix read-out.
- int EXTBLAS_dusm_get (const blas_sparse_matrix A, double ∗value, int ∗row, int ∗col)

  Retrieves a sparse matrix entry.
- int EXTBLAS_dusm_nz (const blas_sparse_matrix A, int ∗nz)

  Retrieves the number of nonzeroes in a given, finalised, sparse matrix.
- int EXTBLAS_dusm_open (const blas_sparse_matrix A)

  Opens a given sparse matrix for read-out.
- int EXTBLAS_dusmsm (const enum blas_trans_type transa, const double alpha, const blas_sparse_matrix A, const enum blas_trans_type transb, const blas_sparse_matrix B, blas_sparse_matrix C)

  Performs sparse matrix–sparse matrix multiplication.
- int EXTBLAS_dusmsv (const enum blas_trans_type transa, const double alpha, const blas_sparse_matrix A, const extblas_sparse_vector x, extblas_sparse_vector y)

  Performs sparse matrix–sparse vector multiplication.
- int EXTBLAS_free ()

  This function is an implementation-specific extension of SparseBLAS that clears any buffer memory that preceding SparseBLAS operations may have created and used.

10.89.1 Detailed Description

This is the ALP implementation of a subset of the NIST Sparse BLAS standard.

While the API is standardised, this header makes some implementation-specific extensions.

10.89.2 Typedef Documentation

10.89.2.1 blas_sparse_matrix

```
typedef void* blas_sparse_matrix
```

A sparse matrix.

See the SparseBLAS paper for the full specification.

10.89.3 Enumeration Type Documentation

10.89.3.1 blas_order_type

```
enum blas_order_type
```

The supported dense storages.

See the SparseBLAS paper for the full specification.

10.89.3.2 blas_trans_type

```
enum blas_trans_type
```

The possible transposition types.

See the SparseBLAS paper for the full specification.

This implementation at present does not support `blas_conj_trans`.

10.89.4 Function Documentation

10.89.4.1 BLAS_duscr_begin()

```
blas_sparse_matrix BLAS_duscr_begin (
             const int m,
             const int n )
```

Creates a handle to a new / empty sparse matrix.

A call to this function must always be paired with one to

- BLAS_duscr_end

See the SparseBLAS paper for the full specification.

10.89.4.2 BLAS_duscr_end()

```
int BLAS_duscr_end (
             blas_sparse_matrix A )
```

Signals that the matrix *A* can now be finalised – all contents have been added.

See the SparseBLAS paper for the full specification.

10.89.4.3 BLAS_duscr_insert_col()

```
int BLAS_duscr_insert_col (
             blas_sparse_matrix A,
             const int j,
             const int nnz,
             const double * vals,
             const int * rows )
```

Inserts a column into *A*.

See the SparseBLAS paper for the full specification.

10.89.4.4 BLAS_duscr_insert_entries()

```
int BLAS_duscr_insert_entries (
             blas_sparse_matrix A,
             const int nnz,
             const double * vals,
             const int * rows,
             const int * cols )
```

Inserts a block of entries into *A*.

See the SparseBLAS paper for the full specification.

10.89.4.5 BLAS_duscr_insert_entry()

```
int BLAS_duscr_insert_entry (
             blas_sparse_matrix A,
             const double val,
             const int row,
             const int col )
```

Inserts a single nonzero entry into *A*.

See the SparseBLAS paper for the full specification.

10.89.4.6 BLAS_duscr_insert_row()

```
int BLAS_duscr_insert_row (
             blas_sparse_matrix A,
             const int i,
             const int nnz,
             const double * vals,
             const int * cols )
```

Inserts a row into *A*.

See the SparseBLAS paper for the full specification.

10.89.4.7 BLAS_dusmm()

```
int BLAS_dusmm (
             const enum blas_order_type order,
             const enum blas_trans_type transa,
             const int nrhs,
             const double alpha,
             const blas_sparse_matrix A,
             const double * B,
             const int ldb,
             const double * C,
             const int ldc )
```

Sparse matrix–dense matrix multiplication.

This function computes one of

- $C \rightarrow \alpha AB + C$
- $C \rightarrow \alpha A^T B + C$

See the SparseBLAS paper for the full specification.

10.89.4.8 BLAS_dusmv()

```
int BLAS_dusmv (
              const enum blas_trans_type transa,
              const double alpha,
              const blas_sparse_matrix A,
              const double *const x,
              int incx,
              double *const y,
              const int incy )
```

Sparse matrix–dense vector multiplication.

This function computes one of

- $y \rightarrow \alpha Ax + y$
- $y \rightarrow \alpha A^T x + y$

See the SparseBLAS paper for the full specification.

10.89.4.9 BLAS_usds()

```
int BLAS_usds (
              blas_sparse_matrix A )
```

Frees a given matrix.

See the SparseBLAS paper for the full specification.

10.89.4.10 EXTBLAS_dusm_clear()

```
int EXTBLAS_dusm_clear (
              blas_sparse_matrix A )
```

Removes all entries from a finalised sparse matrix.

Parameters

in,out	*A*	The matrix to clear.

Returns

0 If *A* was successfully cleared.

Any other integer in case of error, which brings *A* into an undefined state.

This is an implementation-specific extension.

10.89.4.11 EXTBLAS_dusm_close()

```
int EXTBLAS_dusm_close (
            const blas_sparse_matrix A )
```

Closes a sparse matrix read-out.

Parameters

in	*A*	The matrix which is in a read-out state.

Returns

0 If *A* is successfully returned to a finalised state.

Any other integer in case of error, which brings *A* to an undefined state.

This is an implementation-specific extension.

10.89.4.12 EXTBLAS_dusm_get()

```
int EXTBLAS_dusm_get (
            const blas_sparse_matrix A,
            double * value,
            int * row,
            int * col )
```

Retrieves a sparse matrix entry.

Each call to this function will retrieve a new entry. The order in which entries are returned is unspecified.

Parameters

in	*A*	The matrix to retrieve an entry of.

The given matrix must be opened for read-out, and must not have been closed in the mean time.

Parameters

out	*value*	The value of the retrieved nonzero.
out	*row*	The row coordinate of the retrieved nonzero.
out	*col*	The column coordinate of the retrieved nonzero.

Returns

0 If a nonzero was successfully returned and a next value is not available; i.e., the read-out has completed. When this is returned, *A* will no longer be a legal argument for a call to this function.

1 If a nonzero was successfully returned and a next nonzero is available.

Any other integer in case of error.

In case of error, the output memory areas pointed to by *value*, *row*, and *col* will remain untouched. Furthermore, *A* will no longer be a legal argument for a call to this function.

This is an implementation-specific extension.

10.89.4.13 EXTBLAS_dusm_nz()

```
int EXTBLAS_dusm_nz (
            const blas_sparse_matrix A,
            int * nz )
```

Retrieves the number of nonzeroes in a given, finalised, sparse matrix.

Parameters

in	*A*	The matrix to return the number of nonzeroes of.
out	*nz*	Where to store the number of nonzeroes.

Returns

0 If the function call is successful.

Any other value on error, in which case *nz* will remain untouched.

This is an implementation-specific extension.

10.89.4.14 EXTBLAS_dusm_open()

```
int EXTBLAS_dusm_open (
            const blas_sparse_matrix A )
```

Opens a given sparse matrix for read-out.

Parameters

in	*A*	The matrix to read out.

Returns

0 If the call was successful.

Any other value if it was not, in which case the state of *A* shall remain unchanged.

After a successful call to this function, *A* moves into a read-out state. This means *A* shall only be a valid argument for calls to EXTBLAS_dusm_get and EXTBLAS_dusm_close.

This is an implementation-specific extension.

10.89.4.15 EXTBLAS_dusmsm()

```
int EXTBLAS_dusmsm (
              const enum blas_trans_type transa,
              const double alpha,
              const blas_sparse_matrix A,
              const enum blas_trans_type transb,
              const blas_sparse_matrix B,
              blas_sparse_matrix C )
```

Performs sparse matrix–sparse matrix multiplication.

This function is an implementation-specific extension of SparseBLAS that performs one of

- $C \to \alpha AB + C$,
- $C \to \alpha A^T B + C$,
- $C \to \alpha AB^T + C$, or
- $C \to \alpha A^T B^T + C$.

Parameters

in	*transa*	The requested transposition of *A*.
in	*alpha*	The scalar with which to element-wise multiply the result of AB.
in	*A*	The left-hand input matrix A.
in	*transb*	The requested transposition of *B*.
in	*B*	The right-hand input matrix B.

Parameters

in,out	*C*	The output matrix C into which the result of the matrix–matrix multiplication is added.

Returns

0 If the multiplication has completed successfully.

Any other integer on error, in which case the contents of all arguments to this function shall remain unmodified.

10.89.4.16 EXTBLAS_dusmsv()

```
int EXTBLAS_dusmsv (
             const enum blas_trans_type transa,
             const double alpha,
             const blas_sparse_matrix A,
             const extblas_sparse_vector x,
             extblas_sparse_vector y )
```

Performs sparse matrix–sparse vector multiplication.

This function is an implementation-specific extension of SparseBLAS that performs one of

- $y \rightarrow \alpha Ax + y$, or
- $y \rightarrow \alpha A^T x + y$.

Parameters

in	*transa*	The requested transposition of A.

Parameters

in	*alpha*	The scalar with which to element-wise multiply the result of the matrix–vector multiplication (prior to addition to y).
in	*A*	The matrix A with which to multiply *x*.
in	*x*	The vector x with which to multiply *A*.
in,out	*y*	The output vector y into which the result of the matrix–vector multiplication is added.

Returns

0 If the requested operation completed successfully.

Any other integer in case of error. If returned, all arguments to the call to this function shall remain unmodified.

10.89.4.17 EXTBLAS_free()

```
int EXTBLAS_free ( )
```

This function is an implementation-specific extension of SparseBLAS that clears any buffer memory that preceding SparseBLAS operations may have created and used.

Returns

0 On success.

Any other integer on failure, in which case the ALP/SparseBLAS implementation enters an undefined state.

10.90 blas_sparse.h

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17
26 #ifndef _H_ALP_SPARSEBLAS_NIST
27 #define _H_ALP_SPARSEBLAS_NIST
28
29 #include "blas_sparse_vec.h"
30
31 #ifdef __cplusplus
32 extern "C" {
33 #endif
34
42 enum blas_trans_type {
43     blas_no_trans = 0,
44     blas_trans,
45     blas_conj_trans
46 };
47
53 enum blas_order_type {
54     blas_rowmajor,
55     blas_colmajor
56 };
57
67 typedef void * blas_sparse_matrix;
68
77 blas_sparse_matrix BLAS_duscr_begin( const int m, const int n );
78
84 int BLAS_duscr_insert_entry(
85     blas_sparse_matrix A,
86     const double val,
87     const int row, const int col
88 );
89
95 int BLAS_duscr_insert_entries(
96     blas_sparse_matrix A,
97     const int nnz,
98     const double * vals, const int * rows, const int * cols
99 );
100
106 int BLAS_duscr_insert_col(
107     blas_sparse_matrix A,
108     const int j, const int nnz,
109     const double * vals, const int * rows
110 );
111
117 int BLAS_duscr_insert_row(
```

```
118     blas_sparse_matrix A,
119     const int i, const int nnz,
120     const double * vals, const int * cols
121 );
122
129 int BLAS_duscr_end( blas_sparse_matrix A );
130
136 int BLAS_usds( blas_sparse_matrix A );
137
147 int BLAS_dusmv(
148     const enum blas_trans_type transa,
149     const double alpha, const blas_sparse_matrix A,
150     const double * const x, int incx,
151     double * const y, const int incy
152 );
153
163 int BLAS_dusmm(
164     const enum blas_order_type order,
165     const enum blas_trans_type transa,
166     const int nrhs,
167     const double alpha, const blas_sparse_matrix A,
168     const double * B, const int ldb,
169     const double * C, const int ldc
170 );
171
193 int EXTBLAS_dusmsv(
194     const enum blas_trans_type transa,
195     const double alpha, const blas_sparse_matrix A,
196     const extblas_sparse_vector x,
197     extblas_sparse_vector y
198 );
199
223 int EXTBLAS_dusmsm(
224     const enum blas_trans_type transa,
225     const double alpha, const blas_sparse_matrix A,
226     const enum blas_trans_type transb, const blas_sparse_matrix B,
227     blas_sparse_matrix C
228 );
229
242 int EXTBLAS_dusm_nz( const blas_sparse_matrix A, int * nz );
243
259 int EXTBLAS_dusm_open( const blas_sparse_matrix A );
260
290 int EXTBLAS_dusm_get(
291     const blas_sparse_matrix A,
292     double * value, int * row, int * col
293 );
294
306 int EXTBLAS_dusm_close( const blas_sparse_matrix A );
307
319 int EXTBLAS_dusm_clear( blas_sparse_matrix A );
320
330 int EXTBLAS_free();
331
332 #ifdef __cplusplus
333 } // end extern "C"
334 #endif
335
336 #endif // end _H_ALP_SPARSEBLAS_NIST
337
```

10.91 blas_sparse_vec.h File Reference

This is an ALP-specific extension to the NIST Sparse BLAS standard, which the ALP libsparseblas transition path also introduces to the de-facto spblas standard.

Typedefs

- typedef void * extblas_sparse_vector

  A sparse vector.

Functions

- extblas_sparse_vector EXTBLAS_dusv_begin (const int n)
  Creates a handle to a new sparse vector that holds no entries.
- int EXTBLAS_dusv_clear (extblas_sparse_vector x)
  Removes all entries from a finalised sparse vector.
- int EXTBLAS_dusv_close (const extblas_sparse_vector x)
  Closes a sparse vector read-out.
- int EXTBLAS_dusv_end (extblas_sparse_vector x)
  Signals the end of sparse vector construction, making the given vector ready for use.
- int EXTBLAS_dusv_get (const extblas_sparse_vector x, double ∗const val, int ∗const ind)
  Retrieves a sparse vector entry.
- int EXTBLAS_dusv_insert_entry (extblas_sparse_vector x, const double val, const int index)
  Inserts a new nonzero entry into a sparse vector that is under construction.
- int EXTBLAS_dusv_nz (const extblas_sparse_vector x, int ∗nz)
  Retrieves the number of nonzeroes in a given finalised sparse vector.
- int EXTBLAS_dusv_open (const extblas_sparse_vector x)
  Opens a sparse vector for read-out.
- int EXTBLAS_dusvds (extblas_sparse_vector x)
  Destroys the given sparse vector.

10.91.1 Detailed Description

This is an ALP-specific extension to the NIST Sparse BLAS standard, which the ALP libsparseblas transition path also introduces to the de-facto spblas standard.

10.91.2 Typedef Documentation

10.91.2.1 extblas_sparse_vector

```
typedef void* extblas_sparse_vector
```

A sparse vector.

This is an implementation-specific extension.

10.91.3 Function Documentation

10.91.3.1 EXTBLAS_dusv_begin()

```
extblas_sparse_vector EXTBLAS_dusv_begin (
            const int n )
```

Creates a handle to a new sparse vector that holds no entries.

This is an implementation-specific extension.

Parameters

in	*n*	The returned vector size.

Returns

An extblas_sparse_vector that is under construction.

10.91.3.2 EXTBLAS_dusv_clear()

```
int EXTBLAS_dusv_clear (
            extblas_sparse_vector x )
```

Removes all entries from a finalised sparse vector.

Parameters

in	*x*	The vector to clear.

Returns

0 If *x* was successfully cleared.

Any other integer in case of error, which brings *x* into an undefined state.

This is an implementation-specific extension.

10.91.3.3 EXTBLAS_dusv_close()

```
int EXTBLAS_dusv_close (
            const extblas_sparse_vector x )
```

Closes a sparse vector read-out.

Parameters

in	*x*	The vector which is in a read-out state.

Returns

0 If *x* is successfully returned to a finalised state.

Any other integer in case of error, which brings *A* to an undefined state.

This is an implementation-specific extension.

10.91.3.4 EXTBLAS_dusv_end()

```
int EXTBLAS_dusv_end (
            extblas_sparse_vector x )
```

Signals the end of sparse vector construction, making the given vector ready for use.

Parameters

in,out	*x*	The sparse vector that is under construction.

Returns

0 If *x* has successfully been moved to a finalised state.

Any other integer if the call was unsuccessful, in which case the state of *x* becomes undefined.

This is an implementation-specific extension.

10.91.3.5 EXTBLAS_dusv_get()

```
int EXTBLAS_dusv_get (
            const extblas_sparse_vector x,
            double *const val,
            int *const ind )
```

Retrieves a sparse vector entry.

Each call to this function will retrieve a new entry. The order in which entries are returned is unspecified.

Parameters

in	*x*	The vector to retrieve an entry of.

The given vector must be opened for read-out, and must not have been closed in the mean time.

Parameters

out	*val*	The value of the retrieved nonzero.
out	*ind*	The index of the retrieved nonzero value.

Returns

0 If a nonzero was successfully returned but a next value is not available; i.e., the read-out has completed. When this is returned, *x* will no longer be a legal argument for a call to this function.

1 If a value was successfully returned and a next nonzero is available.

Any other integer in case of error.

In case of error, the output memory areas pointed to by *val* and *ind* shall remain untouched. Furthermore, *x* will no longer be a valid argument for a call to this function.

This is an implementation-specific extension.

10.91.3.6 EXTBLAS_dusv_insert_entry()

```
int EXTBLAS_dusv_insert_entry (
            extblas_sparse_vector x,
            const double val,
            const int index )
```

Inserts a new nonzero entry into a sparse vector that is under construction.

Parameters

in,out	*x*	The sparse vector to which to add a nonzero.
in	*val*	The nonzero to add to *x*.
in	*index*	The nonzero coordinate.

The value *index* must be smaller than the size of the vector *x* as given during the call to EXTBLAS_dusv_begin that returned *x*.

Returns

0 If *x* has successfully ingested the given nonzero.

Any other integer on error, in which case the state of *x* shall become undefined.

This is an implementation-specific extension.

10.91.3.7 EXTBLAS_dusv_nz()

```
int EXTBLAS_dusv_nz (
            const extblas_sparse_vector x,
            int * nz )
```

Retrieves the number of nonzeroes in a given finalised sparse vector.

Parameters

in	*x*	The vector of which to return the number of nonzeroes.
out	*nz*	Where to store the number of nonzeroes in a given sparse vector.

Returns

0 If the call was successful and *nz* was set.

Any other integer if the call was unsuccessful, in which case *nz* shall remain untouched.

This is an implementation-specific extension.

10.91.3.8 EXTBLAS_dusv_open()

```
int EXTBLAS_dusv_open (
            const extblas_sparse_vector x )
```

Opens a sparse vector for read-out.

Parameters

in	*x*	The vector to read out.

Returns

0 If the call was successful.

Any other integer indicating an error, in which case the state of *x* shall remain unchanged.

After a successful call to this function, *x* moves into a read-out state. This means *x* shall only be a valid argument for calls to EXTBLAS_dusv_get and EXTBLAS_dusv_close.

This is an implementation-specific extension.

10.91.3.9 EXTBLAS_dusvds()

```
int EXTBLAS_dusvds (
            extblas_sparse_vector x )
```

Destroys the given sparse vector.

Parameters

in	*x*	The finalised sparse vector to destroy.

Returns

0 If the call was successful, after which *x* should no longer be used unless it is overwritten by a call to EXTBLAS_dusv_begin.

Any other integer if the call was unsuccessful, in which case the state of *x* becomes undefined.

This is an implementation-specific extension.

10.92 blas_sparse_vec.h

Go to the documentation of this file.

```
1
2 /*
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
```

```
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10 *
11 * Unless required by applicable law or agreed to in writing, software
12 * distributed under the License is distributed on an "AS IS" BASIS,
13 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 * See the License for the specific language governing permissions and
15 * limitations under the License.
16 */
17
26 #ifndef _H_ALP_SPARSEBLAS_EXT_VEC
27 #define _H_ALP_SPARSEBLAS_EXT_VEC
28
29 #ifdef __cplusplus
30 extern "C" {
31 #endif
32
34 typedef void * extblas_sparse_vector;
35
45 extblas_sparse_vector EXTBLAS_dusv_begin( const int n );
46
63 int EXTBLAS_dusv_insert_entry(
64     extblas_sparse_vector x,
65     const double val,
66     const int index
67 );
68
81 int EXTBLAS_dusv_end( extblas_sparse_vector x );
82
95 int EXTBLAS_dusvds( extblas_sparse_vector x );
96
110 int EXTBLAS_dusv_nz( const extblas_sparse_vector x, int * nz );
111
127 int EXTBLAS_dusv_open( const extblas_sparse_vector x );
128
157 int EXTBLAS_dusv_get(
158     const extblas_sparse_vector x,
159     double * const val, int * const ind
160 );
161
173 int EXTBLAS_dusv_close( const extblas_sparse_vector x );
174
186 int EXTBLAS_dusv_clear( extblas_sparse_vector x );
187
188 #ifdef __cplusplus
189 } // end extern "C"
190 #endif
191
192 #endif // end '_H_ALP_SPARSEBLAS_EXT_VEC'
193
```

10.93 spblas.h File Reference

This is the ALP implementation of a subset of the de-facto *_spblas.h Sparse BLAS standard.

Functions

- void extspblas_dcsrmultsv (const char *trans, const int *request, const int *m, const int *n, const double *a, const int *ja, const int *ia, const extblas_sparse_vector x, extblas_sparse_vector y)

  Performs sparse matrix–sparse vector multiplication.
- void **extspblas_free** ()

  An extension that frees any buffers the ALP/GraphBLAS-generated SparseBLAS library may have allocated.
- void spblas_dcsrgemv (const char *transa, const int *m, const double *a, const int *ia, const int *ja, const double *x, double *y)

  Performs sparse matrix–vector multiplication.
- void spblas_dcsrmm (const char *transa, const int *m, const int *n, const int *k, const double *alpha, const char *matdescra, const double *val, const int *indx, const int *pntrb, const int *pntre, const double *b, const int *ldb, const double *beta, double *c, const int *ldc)

Computes a variant of $C \rightarrow \alpha AB + \beta C$.

- void spblas_dcsrmultcsr (const char *trans, const int *request, const int *sort, const int *m, const int *n, const int *k, double *a, int *ja, int *ia, double *b, int *jb, int *ib, double *c, int *jc, int *ic, const int *nzmax, int *info)

  Computes $C \rightarrow AB$ or $C \rightarrow A^T B$, where all matrices are sparse and employ the Compressed Row Storage (CRS).

10.93.1 Detailed Description

This is the ALP implementation of a subset of the de-facto *_spblas.h Sparse BLAS standard.

This implementation uses the spblas_ prefix; e.g., spblas_dcsrgemv.

All functions defined have `void` return types. This implies two important factors:

1. when breaking the contract defined in the API, undefined behaviour will occur.
2. this API hence does not permit the graceful handling of any errors that ALP would normally recover gracefully from, such as, but not limited to, the detection of dimension mismatches.

10.93.2 Function Documentation

10.93.2.1 extspblas_dcsrmultsv()

```
void extspblas_dcsrmultsv (
            const char * trans,
            const int * request,
            const int * m,
            const int * n,
            const double * a,
            const int * ja,
            const int * ia,
            const extblas_sparse_vector x,
            extblas_sparse_vector y )
```

Performs sparse matrix–sparse vector multiplication.

This extension performs one of

1. $y \rightarrow y + \alpha Ax$, or
2. $y \rightarrow y + \alpha A^T x$.

Here, A is assumed in Compressed Row Storage (CRS), while x and y are assumed to be using the extblas_sparse_vector extension.

This API follows loosely that of spblas_dcsrmultcsr.

Parameters

in	*trans*	Either 'N' or 'T', indicating whether A is to be transposed. The Hermitian operator on *A* is currently not supported; if required, please submit a ticket.

Parameters

in	*request*	A pointer to an integer that reads either 0 or 1 0: the output vector is guaranteed to have sufficient capacity to hold the output of the computation. 1: a symbolic phase will be executed that only modifies the capacity of the output vector so that it is guaranteed to be able to hold the output of the requested computation.

Parameters

in	*m,n*	Pointers to integers equal to m, n.
in	*a*	The value array of the nonzeroes in A.
in	*ja*	The column indices of the nonzeroes in A.
in	*ia*	The row offset arrays of the nonzeroes in A.
in	*x*	The sparse input vector.
out	*y*	The sparse output vector.

This is an ALP implementation-specific extension.

10.93.2.2 spblas_dcsrgemv()

```
void spblas_dcsrgemv (
            const char * transa,
            const int * m,
            const double * a,
            const int * ia,
            const int * ja,
            const double * x,
            double * y )
```

Performs sparse matrix–vector multiplication.

This function computes one of

- $y \rightarrow Ax$, or
- $y \rightarrow A^T x$.

The matrix A is $m \times n$ and holds k nonzeroes, and is assumed to be stored in Compressed Row Storage (CRS).

Parameters

in	*transa*	Either 'N' or 'T' for transposed ('T') or not ('N').
in	*m*	The row size of A.
in	*a*	The nonzero value array of A of size k.
in	*ia*	The row offset array of A of size $m+1$.
in	*ja*	The column indices of nonzeroes of A. Must be of size k.
in	*x*	The dense input vector x of length n.
out	*y*	The dense output vector y of length m.

All memory regions must be pre-allocated and initialised.

10.93.2.3 spblas_dcsrmm()

```
void spblas_dcsrmm (
            const char * transa,
            const int * m,
            const int * n,
            const int * k,
            const double * alpha,
            const char * matdescra,
            const double * val,
            const int * indx,
            const int * pntrb,
            const int * pntre,
            const double * b,
            const int * ldb,
            const double * beta,
            double * c,
            const int * ldc )
```

Computes a variant of $C \rightarrow \alpha AB + \beta C$.

The matrix A is sparse and employs the Compressed Row Storage (CRS). The matrices B, C are dense. A has size $m \times k$, B is $k \times n$ and C is $m \times n$.

Parameters

in	*transa*	Either 'N' or 'T'.
in	*m,n,k*	Pointers to integers that equal m, n, k, resp.
in	*alpha*	Pointer to the scalar α.

Parameters

in	*matdescra*	Has several entries. Going from first to last: Either 'G', 'S', 'H', 'T', 'A', or 'D' (similar to Matrix↩ Market) Either 'L' or 'U', in the case of 'T' (triangular) Either 'N' or 'U' for the diagonal type Either 'F' or 'C' (one or zero based indexing)
in	*val*	The values of the nonzeroes in A.
in	*indx*	The column index of the nonzeroes in A.

Parameters

in	*pntrb*	The Compressed Row Storage (CRS) row start array.
in	*pntre*	The array *pntrb* shifted by one.
in	*b*	Pointer to the values of B.
in	*ldb*	Leading dimension of *b*. If in row-major format, this should be n. If in column-major format, this should be k.
in	*beta*	Pointer to the scalar β.
in	*c*	Pointer to the values of C.

Parameters

in	*ldc*	Leading dimension of *c*. If in row-major format, this should be n. If in column-major format, this should be m.

10.93.2.4 spblas_dcsrmultcsr()

```
void spblas_dcsrmultcsr (
             const char * trans,
             const int * request,
             const int * sort,
             const int * m,
             const int * n,
             const int * k,
             double * a,
             int * ja,
             int * ia,
             double * b,
             int * jb,
             int * ib,
             double * c,
             int * jc,
             int * ic,
             const int * nzmax,
             int * info )
```

Computes $C \rightarrow AB$ or $C \rightarrow A^T B$, where all matrices are sparse and employ the Compressed Row Storage (CRS).

The matrix C is $m \times n$, the matrix A is $m \times k$, and the matrix B is $k \times n$.

Parameters

in	*trans*	Either 'N' or 'T', indicating whether A is to be transposed. The Hermitian operator on *A* is currently not supported; if required, please submit a ticket.

Parameters

in	*request*	A pointer to an integer that reads either 0, 1, or 2. 0: the output memory area has been pre-allocated and is guaranteed sufficient for storing the output 1: a symbolic phase will be executed that only modifies the row offset array *ic*. This array must have been pre-allocated and of sufficient size ($m+1$). 2: assumes 1 has executed prior to this call

Parameters

in	*sort*	A pointer to an integer value of 7. All other values are not supported by this interface. If you require it, please submit a ticket.
in	*m,n,k*	Pointers to the integer sizes of *A*, *B*, and *C*.
in	*a*	The value array of nonzeroes in *A*.
in	*ja*	The column index array of nonzeroes in *A*.
in	*ia*	The row offset array of nonzeroes in *A*.

Parameters

in	*b,ib,jb*	Similar for the nonzeroes in *B*.
out	*c,ic,jc*	Similar for the nonzeroes in *C*. For these parameters depending on *request* there are various assumptions on capacity and, for *ic*, contents.
in	*nzmax*	A pointer to an integer that holds the capacity of *c* and *jc*.

Parameters

out	*info*	The integer pointed to will be set to 0 if the call was successful, -1 if the routine only computed the required size of *c* and *jc* (stored in *ic*), and any positive integer when computation has proceeded successfully until (but not including) the returned integer.

10.94 spblas.h

Go to the documentation of this file.

```
1
2 /*
```

```
3  *   Copyright 2021 Huawei Technologies Co., Ltd.
4  *
5  * Licensed under the Apache License, Version 2.0 (the "License");
6  * you may not use this file except in compliance with the License.
7  * You may obtain a copy of the License at
8  *
9  *     http://www.apache.org/licenses/LICENSE-2.0
10 *
11 * Unless required by applicable law or agreed to in writing, software
12 * distributed under the License is distributed on an "AS IS" BASIS,
13 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 * See the License for the specific language governing permissions and
15 * limitations under the License.
16 */
17
34 #ifndef _H_ALP_SPBLAS
35 #define _H_ALP_SPBLAS
36
37 #include "blas_sparse_vec.h"
38
39 #ifdef __cplusplus
40 extern "C" {
41 #endif
42
64 void spblas_dcsrgemv(
65     const char * transa,
66     const int * m,
67     const double * a, const int * ia, const int * ja,
68     const double * x,
69     double * y
70 );
71
101 void spblas_dcsrmm(
102     const char * transa,
103     const int * m, const int * n, const int * k,
104     const double * alpha,
105     const char * matdescra, const double * val, const int * indx,
106     const int * pntrb, const int * pntre,
107     const double * b, const int * ldb,
108     const double * beta,
109     double * c, const int * ldc
110 );
111
152 void spblas_dcsrmultcsr(
153     const char * trans, const int * request, const int * sort,
154     const int * m, const int * n, const int * k,
155     double * a, int * ja, int * ia,
156     double * b, int * jb, int * ib,
157     double * c, int * jc, int * ic,
158     const int * nzmax, int * info
159 );
160
192 void extspblas_dcsrmultsv(
193     const char * trans, const int * request,
194     const int * m, const int * n,
195     const double * a, const int * ja, const int * ia,
196     const extblas_sparse_vector x,
197     extblas_sparse_vector y
198 );
199
204 void extspblas_free();
205
206 #ifdef __cplusplus
207 } // end extern "C"
208 #endif
209
210 #endif // end _H_ALP_SPBLAS
211
```

Index