ALP User Documentation

0.7.alpha

Generated by Doxygen 1.9.6

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

Enumerator

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 run-time by the number of active vertices.
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.
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 program chosen.
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.

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.

Enumerations

- enum Backend {
  reference , reference_omp , hyperdags , **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.

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.
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.
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.
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 phases it composes with a single user process backend, reference by default.
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 MEMORY

  Memory configuration parameters.
- class SIMD_SIZE

  The SIMD size, in bytes.

Functions

- static constexpr size_t big_memory ()
- static constexpr size_t inner ()
- static constexpr size_t l1_cache_size ()
- static constexpr size_t outer ()

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 inner()

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

Returns

The default number of inner repetitions.

7.7.2.3 l1_cache_size()

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

Returns

the private L1 data cache size, in bytes.

7.7.2.4 outer()

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

Returns

The default number of outer repetitions.

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.
- 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 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 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 input or input is performed.
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.
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 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*.
`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 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 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 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.
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 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 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 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 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 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 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 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 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 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 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 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 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 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 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.
`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 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 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 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_operation.
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 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 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 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 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 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.
`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 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 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 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 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 using any number of vectors of equal length, following the nonzero pattern of the given vector x.

- 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 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*.
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 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 into a given scalar.
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.
`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.
`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 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.
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::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 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.
`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 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.
in	*x*	The left-hand input vector of type *InputType1*. This may be a sparse vector.
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 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 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.
`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 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 *OutputType*. This may be a sparse vector.
in	*mask*	The output mask.
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* 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 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.
in	*mask*	The output mask.
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.
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 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.
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 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 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.
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() [1/14]

```
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 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	*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 scalar inputs, monoid version. A call to this function (with grb::EXECUTE for *phase*) is equivalent to the following code:

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

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() [2/14]

```
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 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, operator version. A call to this function (with grb::EXECUTE for *phase*) is equivalent to the following code:

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

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() [3/14]

```
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.

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 $\odot$.
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() [4/14]

```
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.

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 $\odot$.
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() [5/14]

```
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.

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 $\odot$.
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() [6/14]

```
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.

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 $\odot$.
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() [7/14]

```
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.

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.
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() [8/14]

```
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.

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 $\odot$.
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() [9/14]

```
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.

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

7.11.3.20 eWiseApply() [10/14]

```
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.

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

7.11.3.21 eWiseApply() [11/14]

```
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.

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

7.11.3.22 eWiseApply() [12/14]

```
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.

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

7.11.3.23 eWiseApply() [13/14]

```
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.

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.
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

7.11.3.24 eWiseApply() [14/14]

```
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.

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

7.11.3.25 eWiseLambda()

```
RC eWiseLambda (
            const Func f,
            const Vector< DataType, backend, Coords > & x,
            Args...    )
```

Executes an arbitrary element-wise user-defined function *f* using any number of vectors of equal length, following the nonzero pattern of the given vector *x*.

The user-defined function is passed as a lambda which can capture, at the very least, other instances of type grb::Vector. Use of this function is preferable whenever multiple element-wise operations are requested that use one or more identical input vectors. Performing the computation one after the other in blocking mode would require the same vector to be streamed multiple times, while with this function the operations can be fused explicitly instead.

It shall always be legal to capture non-GraphBLAS objects for read access only. It shall *not* be legal to capture instances of type grb::Matrix for read and/or write access.

If grb::Properties::writableCaptured evaluates true then captured non-GraphBLAS objects can also be written to, not just read from. The captured variable is, however, completely local to the calling user process only– it will not be synchronised between user processes. As a rule of thumb, data-centric ALP/GraphBLAS implementations *cannot* support this and will thus have grb::Properties::writableCaptured evaluate to false. A portable ALP/GraphBLAS algorithm should provide a different code path to handle this case (or not rely on grb::eWiseLambda). When it is legal to write to captured scalar, this function can, e.g., be used to perform reduction-like operations on any number of equally sized input vectors. This would be preferable to a chained number of calls to grb::dot in case where some vectors are shared between subsequent calls, for example; the shared vectors are streamed only once using this lambda- enabled function.

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 end with a 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 a subsequent call to grb::collectives<>::allreduce.

Note

This is an addition to the GraphBLAS C specification. It is alike user-defined operators, monoids, and semirings, except it allows execution on arbitrarily many inputs and arbitrarily many outputs. It is intended for programmers to take control over what is fused when and how. The #grb::nonblocking backend attempts to automate the application of such fusion opportunities without the user's explicit involvement.

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

`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/GraphBLAS vectors can access that element via the operator[]. It is illegal to access any element not at position *i*. The lambda takes only the single parameter *i* of type `const size_t`. Captured scalars will not be globally updated– the user must program this explicitly. Scalars and other non-GraphBLAS containers are always local to their user process.
`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.

In future ALP/GraphBLAS implementations, apart from performing dimension checking, may also require data redistribution in case different vectors may be 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.

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 );
```

The version using the lambdas, however, is expected to execute faster as both *x* and *y* are streamed only once, while the latter code may stream both vectors twice.

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.

Captured scalars will be local to the user process executing the lambda. To retrieve the global dot product, an allreduce must explicitly be called.

See also

Vector::operator[]()

Vector::lambda_reference

7.11.3.26 eWiseMul() [1/8]

```
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 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`	*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::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

7.11.3.27 eWiseMul() [2/8]

```
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 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	*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.
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

7.11.3.28 eWiseMul() [3/8]

```
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 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*.

Parameters

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 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

7.11.3.29 eWiseMul() [4/8]

```
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 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	*y*	The right-hand input vector 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 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

7.11.3.30 eWiseMul() [5/8]

```
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 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

7.11.3.31 eWiseMul() [6/8]

```
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 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.
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

7.11.3.32 eWiseMul() [7/8]

```
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 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 output mask of type *MaskType*.
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.
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

7.11.3.33 eWiseMul() [8/8]

```
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 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.
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

7.11.3.34 foldl() [1/3]

```
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.

Unmasked monoid variant. See masked variant for the full documentation.

7.11.3.35 foldl() [2/3]

```
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.

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

7.11.3.36 foldl() [3/3]

```
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.

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.

7.11.3.37 foldr() [1/2]

```
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.

Masked variant. See the masked, left-to-right variant for the full documentation.

7.11.3.38 foldr() [2/2]

```
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.

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 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

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 *DataType*) at a given index (i, j). Captured GraphBLAS vectors can access corresponding elements via Vector::operator[ ] or Vector::operator(). It is illegal to access any element not at position *i* if the vector length is equal to the row dimension. It is illegal to access any element not at position *j* if the vector length is equal to the column dimension. Vectors of length neither equal to the column or row dimension may *not* be referenced or undefined behaviour will occur. The reference to the matrix nonzero is non *const* and may thus be modified. New nonzeroes may *not* be added through this lambda functionality. The function *f* must have the following signature: `(DataType &nz, const size_t i, const size_t j)`. The GraphBLAS implementation decides which nonzeroes of *A* are dereferenced, and thus also decides the values *i* and *j* the user function is evaluated on.
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 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 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 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 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 assumed.
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*.
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 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 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 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 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 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 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 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 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 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 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 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 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)

7.15 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.

7.15.1 Detailed Description

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

7.15.2 Enumeration Type Documentation

7.15.2.1 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 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 , **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 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 using any number of vectors of equal length, following the nonzero pattern of the given vector x.
- 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 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.
- 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.
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.
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 GraphBLAS IO functions. **Note** The parallel mode in situations where the number of user processes is one, for instance when choosing a sequential or data-centric GraphBLAS implementation, IOMode::parallel is equivalent to IOMode::sequential.

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

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 using the grb::EXECUTE phase shall be successful. Here, an *equivalent call* means that the operation must be called with exactly the same arguments, except for the grb::Phase argument. Here, *subsequent* means that all involved containers are not arguments to any other ALP/GraphBLAS primitives prior to the final call that requests the execute phase. Different from grb::resize, calling operations using the resize phase does *not* modify the contents of output containers, and may only enlargen capacities– not shrink them. **Note** This specification does *not* disallow implementations or backends that perform part of the computation during the resize phase. Any such behaviour is totally optional for implementations and backends. However, any progress made in such manner must remain hidden from the user since output container contents must not be modified by primitives executing a resize phase. A backend must define clear performance semantics for each primitive and for each phase that primitive can be called with. In particular, backends must specify whether system calls such as dynamic memory allocations or frees may occur, and whether primtives operating in a resize phase may return grb::OUTOFMEM.

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 computation is incomplete and the primitive shall return grb::FAILED. Regarding each output container *A*, the following are guaranteed: 1. the capacity of *A* remains unchanged; 2. contains grb::capacity (of *A*) nonzeroes; 3. has nonzeroes at the coordinates where *A* on entry had nonzeroes; 4. has nonzeroes with values equal to those that would have been computed at its coordinates were the call successul; and 5. does not contain all nonzeroes that would have been present in *A* were the call successful (or otherwise grb::SUCCESS would have been returned instead). **Warning** If execution failed, then even though the semantics guarantee valid partial output, there generally is no way to recover the full output without re-initiating the full computation. In other words, this mechanism does not allow for the partial computation to complete the remainder computation using less effort than the full computation would have required. This is the main difference with the grb::EXECUTE phase. **Note** This phase is particularly useful if partial output is still usable and recomputation to generate the full output is not required. A backend must define clear performance semantics for each primitive and for each phase that primitive can be called with. **Warning** The `try` phase is current experimental and *not* broadly supported in the reference implementation.

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 the execute phase. If, instead, the output container capacity was found to be insufficient, then the requested operation may return grb::FAILED, in which case the contents of output containers shall be cleared. **Note** That on failure a primitive called using the execute phase may destroy any pre-existing contents of output containers is a critical difference with the grb::TRY phase. **Warning** When calling ALP/GraphBLAS primitives without specifying a phase explicitly, this execute phase will be assumed by default. A backend must define clear performance semantics for each primitive and for each phase that primitive can be called with. In particular, backends must specify whether system calls such as dynamic memory allocations or frees may occur, and whether primtives operating in a resize phase may return grb::OUTOFMEM. **Note** Typically, implementations and backends are advised to specify no system calls and in particular dynamic memory management calls are allowed as part of an execute phase.

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.
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 code even if not explicitly documented.
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/GraphBLAS 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 run-time; more often than not, this error signals a logical programming error. This error code may only be returned when explicitly documented as such.
OVERLAP	One or more of the GraphBLAS 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 run-time; more often than not, this error signals a logical programming error. This error code may only be returned when explicitly documented as such, but note the deprecation message– any uses of OVERLAP will be replaced with ILLEGAL before v1.0 is released.
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 run-time; more often than not, this error signals the underlying problem is too large to handle with whatever current resources have been assigned to ALP. This error code may only be returned when explicitly documented as such.
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.
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 run-time; more often than not, this error signals a logical programming error. This error code may only be returned when explicitly documented as such; in other words, the specification precisely determines which (combinations of) inputs are illegal.
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 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 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 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 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, 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 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 BiCGstab.

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 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*.
`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 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

SUCCESS If the computation was successful.

MISMATCH If the vector sizes do not match. The output *similarity* is undefined.

ILLEGAL In case *x* is all zero, and/or when *y* is all zero. The output *similarity* is undefined.

PANIC If an unrecoverable error has been encountered. The output as well as the state of ALP/GraphBLAS 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 kmeans_iteration()

```
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.

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
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

8.2.2.5 knn()

```
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.

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.

8.2.2.6 kpp_initialisation()

```
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

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
in	*dist_op*	Coordinatewise distance operator, squared difference by default

8.2.2.7 label()

```
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 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.
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.
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.
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.

8.2.2.8 mpv()

```
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.

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.
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.

8.2.2.9 norm2()

```
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.

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.

8.2.2.10 simple_pagerank()

```
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 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

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 PageRank vector corresponding to *L*.
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 PageRank algorithm.
`in,out`	*pr_nextnext*	Buffer for the PageRank algorithm.
`in,out`	*row_sum*	Buffer for the PageRank 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.
`in`	*conv*	If the difference between two successive iterations, in terms of its one-norm, is less than this value, then the PageRank 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 negative.
`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.

8.2.2.11 sparse_nn_single_inference() [1/2]

```
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 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.
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 GraphChallenge.
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.

8.2.2.12 sparse_nn_single_inference() [2/2]

```
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.

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. 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.

8.2.2.13 spy() [1/3]

```
RC spy (
            grb::Matrix< IOType > & out,
            const grb::Matrix< bool > & in )
```

Specialisation for boolean input matrices *in*.

See grb::algorithms::spy.

8.2.2.14 spy() [2/3]

```
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 Parameters

normalize	If set to true, will not compute a number of mapped nonzeroes, but its inverse instead (one divided by the count). The default value for this template parameter is `false`.

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.

8.2.2.15 spy() [3/3]

```
RC 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 BENCHMARKING

  Benchmarking default configuration parameters.
- class CACHE_LINE_SIZE

  Contains information about the target architecture cache line size.
- class IMPLEMENTATION< BSP1D >

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

9.3.1 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

9.4 argmax< IType, VType > Class Template Reference

The argmax operator on key-value pairs.

`#include <ops.hpp>`

Inherits Operator< internal::argmax< IType, VType > >.

9.4.1 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

9.5 argmin< IType, VType > Class Template Reference

The argmin operator on key-value pairs.

`#include <ops.hpp>`

Inherits Operator< internal::argmin< IType, VType > >.

9.5.1 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

9.6 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.

9.6.1 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

9.6.2 Constructor & Destructor Documentation

9.6.2.1 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.
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

9.6.3 Member Function Documentation

9.6.3.1 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
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

9.6.3.2 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 use rogram to be benchmarked
in	*data_in*	Input data as a raw data blob

Parameters

in	*in_size*	The size, in bytes, of the input data
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

9.6.3.3 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

9.7 BENCHMARKING Class Reference

Benchmarking default configuration parameters.

`#include <config.hpp>`

Static Public Member Functions

- static constexpr size_t inner ()
- static constexpr size_t outer ()

9.7.1 Detailed Description

Benchmarking default configuration parameters.

The documentation for this class was generated from the following file:

- base/config.hpp

9.8 CACHE_LINE_SIZE Class Reference

Contains information about the target architecture cache line size.

`#include <config.hpp>`

9.8.1 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

9.9 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.

9.9.1 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.

9.9.2 Member Function Documentation

9.9.2.1 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: $Ng + N * Operator + l$;

9.9.2.2 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*.
`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.

9.9.2.3 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.

9.9.2.4 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.
`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

9.10 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.

9.10.1 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.

9.10.2 Member Function Documentation

9.10.2.1 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`.

9.10.2.2 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.
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

9.11 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.

9.11.1 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.

9.11.2 Member Function Documentation

9.11.2.1 operator"!=()

```
bool operator!= (
            const const_iterator & other ) const  [inline]
```

Returns

The negation of operator==()

9.11.2.2 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.

9.11.2.3 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.

9.11.2.4 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

9.12 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.

9.12.1 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.

9.12.2 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 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.

9.12.2.3 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.

The documentation for this class was generated from the following file:

- vector.hpp

9.13 ConnectedComponents< VertexIDType >::Data Struct Reference

This vertex-centric Connected Components algorithm does not require any algorithm parameters.

`#include <pregel_connected_components.hpp>`

9.13.1 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

9.14 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.

9.14.1 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

9.15 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 > >.

9.15.1 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

9.16 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 > >.

9.16.1 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

9.17 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 > >.

9.17.1 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

9.18 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 > >.

9.18.1 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

9.19 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.

9.19.1 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

9.20 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

9.20.1 Detailed Description

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

9.20.2 Member Function Documentation

9.20.2.1 defaultAllocMode()

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

Returns

The default allocation strategy for private memory segments.

9.20.2.2 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 a single application.

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

9.21 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.

9.21.1 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 documentation for this class was generated from the following file:

- reference/config.hpp

9.22 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.

9.22.1 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.

9.22.2 Member Function Documentation

9.22.2.1 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

9.23 infinity< D > Class Template Reference

Standard identity for the minimum operator.

```
#include <identities.hpp>
```

Static Public Member Functions

- static constexpr D value ()

9.23.1 Detailed Description

template<typename D>
class grb::identities::infinity< D >

Standard identity for the minimum operator.

9.23.2 Member Function Documentation

9.23.2.1 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

9.24 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.

9.24.1 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

9.25 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.

9.25.1 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

9.26 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.

9.26.1 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

9.27 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.

9.27.1 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

9.28 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.

9.28.1 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

9.29 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.

9.29.1 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

9.30 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.

9.30.1 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

9.31 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.

9.31.1 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

9.32 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.

9.32.1 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.

9.32.2 Constructor & Destructor Documentation

9.32.2.1 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

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.
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'.
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.

9.32.3 Member Function Documentation

9.32.3.1 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.

9.32.3.2 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.
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.

9.32.3.3 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

9.33 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 > >.

9.33.1 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

9.34 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 > >.

9.34.1 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

9.35 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 > >.

9.35.1 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

9.36 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 ()

9.36.1 Detailed Description

template<typename D>
class grb::identities::logical_false< D >

Standard identitity for the logical or operator.

See also

operators::logical_or.

9.36.2 Member Function Documentation

9.36.2.1 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

9.37 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 > >.

9.37.1 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

9.38 logical_true< D > Class Template Reference

Standard identity for the logical AND operator.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.38.1 Detailed Description

template<typename D>
class grb::identities::logical_true< D >

Standard identity for the logical AND operator.

See also

operators::logical_and.

9.38.2 Member Function Documentation

9.38.2.1 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

9.39 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.

9.39.1 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.

9.39.2 Constructor & Destructor Documentation

9.39.2.1 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.

9.39.2.2 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.

9.39.2.3 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.

9.39.2.4 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

9.39.2.5 ~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.

9.39.3 Member Function Documentation

9.39.3.1 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.

9.39.3.2 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.

9.39.3.3 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.

9.39.3.4 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.

9.39.3.5 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

9.40 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 > >.

9.40.1 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

9.41 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 ()

9.41.1 Detailed Description

Memory configuration parameters.

The documentation for this class was generated from the following file:

- base/config.hpp

9.42 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 > >.

9.42.1 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

9.43 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.

9.43.1 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').

9.43.2 Constructor & Destructor Documentation

9.43.2.1 Monoid()

```
Monoid ( )  [inline]
```

Constructor that infers a default operator, given the operator type.

Useful for stateless operators.

9.43.3 Member Function Documentation

9.43.3.1 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.

9.43.3.2 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

9.44 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 > >.

9.44.1 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

9.45 negative_infinity< D > Class Template Reference

Standard identity for the maximum operator.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.45.1 Detailed Description

template<typename D>
class grb::identities::negative_infinity< D >

Standard identity for the maximum operator.

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 max operator, i.e., 'minus infinity'.

The documentation for this class was generated from the following file:

- identities.hpp

9.46 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 > >.

9.46.1 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

9.47 one< D > Class Template Reference

Standard identity for numerical multiplication.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.47.1 Detailed Description

template<typename D>
class grb::identities::one< D >

Standard identity for numerical multiplication.

9.47.2 Member Function Documentation

9.47.2.1 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

9.48 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.

9.48.1 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 vertices have converged.

9.48.2 Member Function Documentation

9.48.2.1 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.
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.

9.48.2.2 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

9.49 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

9.49.1 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.

9.49.2 Constructor & Destructor Documentation

9.49.2.1 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.

9.49.2.2 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.

9.49.2.3 ~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.

9.49.3 Member Function Documentation

9.49.3.1 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.

9.49.3.2 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.

9.49.3.3 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 be cast to *OutputType*.

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.

9.49.3.4 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.

9.49.3.5 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

9.50 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.

9.50.1 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.

9.50.2 Member Function Documentation

9.50.2.1 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

9.51 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.

9.51.1 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.

9.51.2 Constructor & Destructor Documentation

9.51.2.1 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.

9.51.3 Member Function Documentation

9.51.3.1 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.
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.

9.51.3.2 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.

9.51.3.3 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.

9.51.3.4 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

9.52 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.

9.52.1 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.

9.52.2 Member Data Documentation

9.52.2.1 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.

9.52.2.2 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).

9.52.2.3 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

9.53 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 > >.

9.53.1 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

9.54 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 > >.

9.54.1 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

9.55 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 > >.

9.55.1 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

9.56 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.

9.56.1 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').

9.56.2 Member Typedef Documentation

9.56.2.1 D3

```
typedef _OP2::D3 D3
```

The output domain of the multiplicative operator.

The first input domain of the additive operator.

9.56.2.2 D4

```
typedef _OP1::D2 D4
```

The second input domain of the additive operator.

The output domain of the additive operator.

9.56.3 Member Function Documentation

9.56.3.1 getAdditiveMonoid()

```
AdditiveMonoid getAdditiveMonoid ( ) const  [inline]
```

Retrieves the underlying additive monoid.

Returns

The underlying monoid. Any state is copied.

9.56.3.2 getAdditiveOperator()

```
AdditiveOperator getAdditiveOperator ( ) const  [inline]
```

Retrieves the underlying additive operator.

Returns

The underlying operator. Any state is copied.

9.56.3.3 getMultiplicativeMonoid()

```
MultiplicativeMonoid getMultiplicativeMonoid ( ) const  [inline]
```

Retrieves the underlying multiplicative monoid.

Returns

The underlying monoid. Any state is copied.

9.56.3.4 getMultiplicativeOperator()

```
MultiplicativeOperator getMultiplicativeOperator ( ) const  [inline]
```

Retrieves the underlying multiplicative operator.

Returns

The underlying operator. Any state is copied.

9.56.3.5 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 the same default type as getZero().

Returns

The one corresponding to this semiring, cast to the requested domain.

9.56.3.6 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 expression $a_{ij}x_j$ where often the left- hand side is zero.

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

9.57 SIMD_SIZE Class Reference

The SIMD size, in bytes.

`#include <config.hpp>`

9.57.1 Detailed Description

The SIMD size, in bytes.

The documentation for this class was generated from the following file:

- base/config.hpp

9.58 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

9.58.1 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.

9.58.2 Member Function Documentation

9.58.2.1 nprocs()

```
static size_t nprocs ( )  [inline], [static], [noexcept]
```

Returns

The number of user processes in this ALP run.

9.58.2.2 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

9.59 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 > >.

9.59.1 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

9.60 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 > >.

9.60.1 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

9.61 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.

9.61.1 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.

9.61.2 Member Typedef Documentation

9.61.2.1 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

9.61.3 Constructor & Destructor Documentation

9.61.3.1 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.

9.61.3.2 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*.

9.61.3.3 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.

9.61.3.4 ~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.

9.61.4 Member Function Documentation

9.61.4.1 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.

9.61.4.2 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.
`in`	*start*	The iterator to the first element that should be copied into this GraphBLAS vector.
`in`	*end*	Iterator shifted exactly one past the last element that should be copied into this GraphBLAS 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.

9.61.4.3 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.
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*.
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.

9.61.4.4 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.

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*.
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.

9.61.4.5 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.

9.61.4.6 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.

9.61.4.7 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.

9.61.4.8 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.

9.61.4.9 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.

9.61.4.10 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.

9.61.4.11 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.

9.61.4.12 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

9.62 zero< D > Class Template Reference

Standard identity for numerical addition.

`#include <identities.hpp>`

Static Public Member Functions

- static constexpr D value ()

9.62.1 Detailed Description

template<typename D>
class grb::identities::zero< D >

Standard identity for numerical addition.

9.62.2 Member Function Documentation

9.62.2.1 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

9.63 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 > >.

9.63.1 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.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00368 #ifdef __DOXYGEN__
00369
00377 #define _GRB_NO_LIBNUMA
00378
00386 #define _GRB_NO_PINNING
00387
00398 #define _GRB_NO_STDIO
00399
00415 #define _GRB_NO_EXCEPTIONS
00416
00424 #define _GRB_WITH_LPF
00425
00434 #define _GRB_BACKEND reference
00435
00442 #define _GRB_BSP1D_BACKEND
00443
00450 namespace grb {
00451
00455     namespace algorithms {
00456
00460         namespace pregel {}
00461
00462     }
00463
00468     namespace interfaces {}
00469
00470 }
00471
00472 #endif // end "#ifdef __DOXYGEN__"
00473
00474 #ifndef _H_GRAPHBLAS
00475 #define _H_GRAPHBLAS
00476
00477 // load active configuration
00478 #include <graphblas/config.hpp> //defines _GRB_BACKEND and _WITH_BSP
00479
00480 // collects the user-level includes
00481 // the order of these includes matter--
00482 //    do not modify without proper consideration!
00483
00484 // First include all algebraic structures, which have the benefit of not
00485 // depending on anything else
00486 #include <graphblas/ops.hpp>
00487 #include <graphblas/monoid.hpp>
00488 #include <graphblas/semiring.hpp>
00489
00490 // Then include containers. If containers rely on ALP/GraphBLAS primitives that
00491 // are defined as free functions, then container implementations must forward-
00492 // declare those.
00493 #include <graphblas/vector.hpp>
00494 #include <graphblas/matrix.hpp>
00495
00496 // The aforementioned forward declarations must be in sync with the
00497 // declarations of the user primitives defined as free functions in the below.
00498 // The below relies on both algebraic structures/relations as well as container
00499 // definitions. By maintaining the current order, these do not require forward
00500 // declarations.
00501 #include <graphblas/io.hpp>
00502 #include <graphblas/benchmark.hpp>
00503 #include <graphblas/blas0.hpp>
00504 #include <graphblas/blas1.hpp>
00505 #include <graphblas/blas2.hpp>
00506 #include <graphblas/blas3.hpp>
00507 #include <graphblas/collectives.hpp>
00508 #include <graphblas/exec.hpp>
```

```
00509 #include <graphblas/init.hpp>
00510 #include <graphblas/ops.hpp>
00511 #include <graphblas/pinnedvector.hpp>
00512 #include <graphblas/properties.hpp>
00513 #include <graphblas/spmd.hpp>
00514
00515 #ifdef _GRB_WITH_LPF
00516  // collects various BSP utilities
00517  #include <graphblas/bsp/spmd.hpp>
00518 #endif
00519
00520 #endif // end "_H_GRAPHBLAS"
00521
```

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< Input↩ Type > &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.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00035 #ifndef _H_GRB_ALGORITHMS_BICGSTAB
00036 #define _H_GRB_ALGORITHMS_BICGSTAB
00037
00038 #include <graphblas.hpp>
00039
00040 #include <iostream>
00041 #include <type_traits>
00042
00043 #ifdef _DEBUG
00044  #include <cmath> // for sqrt, making the silent assumption that ResidualType
00045                   // is a supported type for it
00046 #endif
00047
00048
00049 namespace grb {
00050
00051     namespace algorithms {
00052
00054         template<
00055             Descriptor descr = descriptors::no_operation,
00056             typename IOType, typename NonzeroType, typename InputType,
00057             typename ResidualType,
00058             class Semiring = Semiring<
00059                 operators::add< InputType, InputType, InputType >,
00060                 operators::mul< IOType, NonzeroType, InputType >,
00061                 identities::zero, identities::one
00062             >,
00063             class Minus = operators::subtract< ResidualType >,
00064             class Divide = operators::divide< ResidualType >
00065         >
00066         RC bicgstab(
00067             grb::Vector< IOType > &x,
00068             const grb::Matrix< NonzeroType > &A,
00069             const grb::Vector< InputType > &b,
00070             const size_t max_iterations,
00071             ResidualType tol,
00072             size_t &iterations,
00073             ResidualType &residual,
00074             Vector< InputType > &r,
00075             Vector< InputType > &rhat,
00076             Vector< InputType > &p,
00077             Vector< InputType > &v,
00078             Vector< InputType > &s,
00079             Vector< InputType > &t,
00080             const Semiring &semiring = Semiring(),
00081             const Minus &minus = Minus(),
00082             const Divide &divide = Divide()
00083         ) {
00084             // static checks
00085             static_assert( !( descr & descriptors::no_casting ) || (
00086                     std::is_same< IOType, NonzeroType >::value &&
00087                     std::is_same< IOType, InputType >::value &&
00088                     std::is_same< IOType, ResidualType >::value
00089                 ), "no_casting descriptor was set but containers with differing domains "
00090                 "were given."
00091             );
00092             static_assert( !( descr & descriptors::no_casting ) || (
00093                     std::is_same< NonzeroType, typename Semiring::D1 >::value &&
00094                     std::is_same< IOType, typename Semiring::D2 >::value &&
00095                     std::is_same< InputType, typename Semiring::D3 >::value &&
00096                     std::is_same< InputType, typename Semiring::D4 >::value
00097                 ), "no_casting descriptor was set, but semiring has incompatible domains "
00098                 "with the given containers."
00099             );
00200             static_assert( !( descr & descriptors::no_casting ) || (
```

```
00201                     std::is_same< InputType, typename Minus::D1 >::value &&
00202                     std::is_same< InputType, typename Minus::D2 >::value &&
00203                     std::is_same< InputType, typename Minus::D3 >::value
00204                 ), "no_casting descriptor was set, but given minus operator has "
00205                 "incompatible domains with the given containers."
00206             );
00207             static_assert( !( descr & descriptors::no_casting ) || (
00208                     std::is_same< ResidualType, typename Divide::D1 >::value &&
00209                     std::is_same< ResidualType, typename Divide::D2 >::value &&
00210                     std::is_same< ResidualType, typename Divide::D3 >::value
00211                 ), "no_casting descriptor was set, but given divide operator has "
00212                 "incompatible domains with the given tolerance type."
00213             );
00214             static_assert( std::is_floating_point< ResidualType >::value,
00215                 "Require floating-point residual type."
00216             );
00217
00218 #ifdef _DEBUG
00219             std::cout « "Entering bicgstab; "
00220                 « "tol = " « tol « ", "
00221                 « "max_iterations = " « max_iterations « "\n";
00222 #endif
00223
00224             // descriptor for indiciating dense computations
00225             constexpr Descriptor dense_descr = descr | descriptors::dense;
00226
00227             // get an alias to zero and one in case 1 and 0 can't cast properly
00228             const ResidualType zero = semiring.template getZero< ResidualType >();
00229             const ResidualType one  = semiring.template getOne< ResidualType >();
00230
00231             // dynamic checks, sizes:
00232             const size_t n = nrows( A );
00233             if( n != ncols( A ) ) {
00234                 return MISMATCH;
00235             }
00236             if( n != size( x ) ) {
00237                 return MISMATCH;
00238             }
00239             if( n != size( b ) ) {
00240                 return MISMATCH;
00241             }
00242             if( n != size( r ) || n != size( rhat ) || n != size( p ) ||
00243                 n != size( p ) || n != size( s ) || n != size( t )
00244             ) {
00245                 return MISMATCH;
00246             }
00247
00248             // dynamic checks, capacity:
00249             if( n != capacity( x ) ) {
00250                 return ILLEGAL;
00251             }
00252             if( n != capacity( r ) || n != capacity( rhat ) || n != capacity( p ) ||
00253                 n != capacity( p ) || n != capacity( s ) || n != capacity( t )
00254             ) {
00255                 return ILLEGAL;
00256             }
00257
00258             // dynamic checks, others:
00259             if( tol <= zero ) {
00260                 return ILLEGAL;
00261             }
00262
00263 #ifdef _DEBUG
00264             std::cout « "\t dynamic run-time error checking passed\n";
00265 #endif
00266
00267             // prelude
00268             ResidualType b_norm_squared = zero;
00269             RC ret = dot< dense_descr >( b_norm_squared, b, b, semiring );
00270             if( ret ) {
00271                 std::cerr « "Error: BiCGstab encountered \"" « toString(ret)
00272                     « "\" during computation of the norm of b\n";
00273                 return ret;
00274             }
00275
00276             // make it so that we do not need to take square roots when detecting
00277             // convergence
00278             tol *= tol;
00279             tol *= b_norm_squared;
00280 #ifdef _DEBUG
00281             std::cout « "Effective squared relative tolerance is " « tol « "\n";
00282 #endif
00283
00284             // ensure that x is structurally dense
00285             if( nnz( x ) != n ) {
00286                 ret = grb::set< descriptors::invert_mask | descriptors::structural >(
00287                     x, x, zero
```

```
00288                 );
00289                 assert( nnz( x ) == n );
00290             }
00291
00292             // compute residual (squared), taking into account that b may be sparse
00293             residual = zero;
00294             ret = ret ? ret : set( t, zero );                                   // t = Ax
00295             ret = ret ? ret : mxv< dense_descr >( t, A, x, semiring );
00296             assert( nnz( t ) == n );
00297             ret = ret ? ret : set( r, zero );                                // r = b - Ax
00298             ret = ret ? ret : foldl( r, b, semiring.getAdditiveMonoid() );
00299             assert( nnz( r ) == n );
00300             ret = ret ? ret : foldl< dense_descr >( r, t, minus );
00301             ret = ret ? ret : dot< dense_descr >( residual, r, r, semiring ); // residual
00302
00303             // check for prelude error
00304             if( ret ) {
00305                 std::cerr « "Error: BiCGstab encountered \"" « toString(ret)
00306                     « "\" during prelude\n";
00307                 return ret;
00308             }
00309
00310             // check if the guess was good enough
00311             if( residual < tol ) {
00312                 return SUCCESS;
00313             }
00314
00315 #ifdef _DEBUG
00316             std::cout « "\t prelude completed\n";
00317 #endif
00318
00319             // start iterations
00320             ret = ret ? ret : set( rhat, r );
00321             ret = ret ? ret : set( p, zero );
00322             ret = ret ? ret : set( v, zero );
00323             ResidualType rho, rho_old, alpha, beta, omega, temp;
00324             rho_old = alpha = omega = one;
00325             iterations = 0;
00326
00327             for( ; ret == SUCCESS && iterations < max_iterations; ++iterations ) {
00328
00329 #ifdef _DEBUG
00330                 std::cout « "\t iteration " « iterations « " starts\n";
00331 #endif
00332
00333                 // rho = ( rhat, r )
00334                 rho = zero;
00335                 ret = ret ? ret : dot< dense_descr >( rho, rhat, r, semiring );
00336 #ifdef _DEBUG
00337                 std::cout « "\t\t rho = " « rho « "\n";
00338 #endif
00339                 if( ret == SUCCESS && rho == zero ) {
00340                     std::cerr « "Error: BiCGstab detects r at iteration " « iterations «
00341                         " is orthogonal to r-hat\n";
00342                     return FAILED;
00343                 }
00344
00345                 // beta = (rho / rho_old) * (alpha / omega)
00346                 ret = ret ? ret : apply( beta, rho, rho_old, divide );
00347                 ret = ret ? ret : apply( temp, alpha, omega, divide );
00348                 ret = ret ? ret : foldl( beta, temp, semiring.getMultiplicativeOperator() );
00349 #ifdef _DEBUG
00350                 std::cout « "\t\t beta = " « beta « "\n";
00351 #endif
00352
00353                 // p = r + beta ( p - omega * v )
00354                 ret = ret ? ret : eWiseLambda(
00355                     [&r,&beta,&p,&v,&omega,&semiring,&minus] (const size_t i) {
00356                         InputType tmp;
00357                         apply( tmp, omega, v[i], semiring.getMultiplicativeOperator() );
00358                         foldl( p[ i ], tmp, minus );
00359                         foldr( beta, p[ i ], semiring.getMultiplicativeOperator() );
00360                         foldr( r[ i ], p[ i ], semiring.getAdditiveOperator() );
00361                     }, v, b
00362                 );
00363
00364                 // v = Ap
00365                 ret = ret ? ret : set( v, zero );
00366                 ret = ret ? ret : mxv< dense_descr >( v, A, p, semiring );
00367
00368                 // alpha = rho / (rhat, v)
00369                 alpha = zero;
00370                 ret = ret ? ret : dot< dense_descr >( alpha, rhat, v, semiring );
00371                 if( alpha == zero ) {
00372                     std::cerr « "Error: BiCGstab detects rhat is orthogonal to v=Ap "
00373                         « "at iteration " « iterations « ".\n";
00374                     return FAILED;
```

```
00375                 }
00376                 ret = ret ? ret : foldr( rho, alpha, divide );
00377 #ifdef _DEBUG
00378                 std::cout « "\t\t alpha = " « alpha « "\n";
00379 #endif
00380
00381                 // x += alpha * p is post-poned to either the pre-stabilisation exit, or
00382                 // after the stabilisation step
00383                 //ret = ret ? ret : eWiseMul( x, alpha, p, semiring );
00384
00385                 // s = r - alpha * v
00386                 {
00387                     ResidualType minus_alpha = zero;
00388                     ret = ret ? ret : foldl( minus_alpha, alpha, minus );
00389                     ret = ret ? ret : set( s, r );
00390                     ret = ret ? ret : eWiseMul< dense_descr >( s, minus_alpha, v, semiring );
00391                 }
00392
00393                 // check residual
00394                 residual = zero;
00395                 ret = ret ? ret : dot< dense_descr >( residual, s, s, semiring );
00396                 assert( residual > zero );
00397 #ifdef _DEBUG
00398                 std::cout « "\t\t running residual, pre-stabilisation: " « sqrt(residual)
00399                     « "\n";
00400 #endif
00401                 if( ret == SUCCESS && residual < tol ) {
00402                     // update result (x += alpha * p) and exit
00403                     ret = eWiseMul< dense_descr >( x, alpha, p, semiring );
00404                     return ret;
00405                 }
00406
00407                 // t = As
00408                 ret = ret ? ret : set( t, zero );
00409                 ret = ret ? ret : mxv< dense_descr >( t, A, s, semiring );
00410
00411                 // omega = (t, s) / (t, t);
00412                 omega = temp = zero;
00413                 ret = ret ? ret : dot< dense_descr >( temp, t, s, semiring );
00414 #ifdef _DEBUG
00415                 std::cout « "\t\t (t, s) = " « temp « "\n";
00416 #endif
00417                 if( ret == SUCCESS && temp == zero ) {
00418                     std::cerr « "Error: BiCGstab detects As at iteration " « iterations «
00419                         " is orthogonal to s\n";
00420                     return FAILED;
00421                 }
00422                 ret = ret ? ret : dot< dense_descr >( omega, t, t, semiring );
00423 #ifdef _DEBUG
00424                 std::cout « "\t\t (t, t) = " « omega « "\n";
00425 #endif
00426                 assert( omega > zero );
00427                 ret = ret ? ret : foldr( temp, omega, divide );
00428 #ifdef _DEBUG
00429                 std::cout « "\t\t omega = " « omega « "\n";
00430 #endif
00431
00432                 // x += alpha * p + omega * s
00433                 ret = ret ? ret : eWiseMul< dense_descr >( x, alpha, p, semiring );
00434                 ret = ret ? ret : eWiseMul< dense_descr >( x, omega, s, semiring );
00435
00436                 // r = s - omega * t
00437                 {
00438                     ResidualType minus_omega = zero;
00439                     ret = ret ? ret : foldl( minus_omega, omega, minus );
00440                     ret = ret ? ret : set( r, s );
00441                     ret = ret ? ret : eWiseMul< dense_descr >( r, minus_omega, t, semiring );
00442                 }
00443
00444                 // check residual
00445                 residual = zero;
00446                 ret = ret ? ret : dot< dense_descr >( residual, r, r, semiring );
00447                 assert( residual > zero );
00448 #ifdef _DEBUG
00449                 std::cout « "\t\t running residual, post-stabilisation: "
00450                     « sqrt(residual) « ". "
00451                     « "Residual squared: " « residual « ".\n";
00452 #endif
00453                 if( ret == SUCCESS ) {
00454                        if( residual < tol ) { return SUCCESS; }
00455
00456                     // go to next iteration
00457                     rho_old = rho;
00458                 }
00459             }
00460
00461             if( ret == SUCCESS ) {
```

```
00462                 // if we are here, then we did not detect convergence
00463                 std::cerr « "Warning: call to BiCGstab did not converge within "
00464                     « max_iterations « " iterations. Squared two-norm of the running "
00465                     « "residual is " « residual « ". "
00466                     « "Target residual squared: " « tol « ".\n";
00467                 return FAILED;
00468             } else {
00469                 // if we are here, we exited due to an ALP error code
00470                 std::cerr « "Error: BiCGstab encountered error \"" « toString(ret)
00471                     « "\" while iterating to " « iterations « ", ";
00472                 if( iterations == max_iterations ) {
00473                     std::cerr « "which also is the maximum number of iterations.\n";
00474                 } else {
00475                     std::cerr « "which is below the maximum number of iterations of "
00476                         « max_iterations « "\n";
00477                 }
00478                 return ret;
00479             }
00480         }
00481
00482     }
00483 }
00484
00485 #endif // end _H_GRB_ALGORITHMS_BICGSTAB
00486
```

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.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00026 #ifndef _H_GRB_ALGORITHMS_CONJUGATE_GRADIENT
00027 #define _H_GRB_ALGORITHMS_CONJUGATE_GRADIENT
00028
00029 #include <cstdio>
00030 #include <complex>
00031
00032 #include <graphblas.hpp>
00033 #include <graphblas/utils/iscomplex.hpp>
00034
00035
00036 namespace grb {
00037
00038     namespace algorithms {
00039
00147         template< Descriptor descr = descriptors::no_operation,
00148             typename IOType,
00149             typename ResidualType,
00150             typename NonzeroType,
00151             typename InputType,
00152             class Ring = Semiring<
00153                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
00154                 grb::identities::zero, grb::identities::one
00155             >,
00156             class Minus = operators::subtract< IOType >,
00157             class Divide = operators::divide< IOType >
00158         >
00159         grb::RC conjugate_gradient(
00160             grb::Vector< IOType > &x,
00161             const grb::Matrix< NonzeroType > &A,
00162             const grb::Vector< InputType > &b,
00163             const size_t max_iterations,
00164             ResidualType tol,
00165             size_t &iterations,
00166             ResidualType &residual,
00167             grb::Vector< IOType > &r,
00168             grb::Vector< IOType > &u,
00169             grb::Vector< IOType > &temp,
00170             const Ring &ring = Ring(),
00171             const Minus &minus = Minus(),
00172             const Divide &divide = Divide()
00173         ) {
00174             // static checks
00175             static_assert( std::is_floating_point< ResidualType >::value,
00176                 "Can only use the CG algorithm with floating-point residual "
00177                 "types." ); // unless some different norm were used: issue #89
00178             static_assert( !( descr & descriptors::no_casting ) || (
00179                     std::is_same< IOType, ResidualType >::value &&
00180                     std::is_same< IOType, NonzeroType >::value &&
00181                     std::is_same< IOType, InputType >::value
00182                 ), "One or more of the provided containers have differing element types "
00183                 "while the no-casting descriptor has been supplied"
00184             );
00185             static_assert( !( descr & descriptors::no_casting ) || (
00186                     std::is_same< NonzeroType, typename Ring::D1 >::value &&
00187                     std::is_same< IOType, typename Ring::D2 >::value &&
00188                     std::is_same< InputType, typename Ring::D3 >::value &&
00189                     std::is_same< InputType, typename Ring::D4 >::value
00190                 ), "no_casting descriptor was set, but semiring has incompatible domains "
00191                 "with the given containers."
00192             );
00193             static_assert( !( descr & descriptors::no_casting ) || (
00194                     std::is_same< InputType, typename Minus::D1 >::value &&
00195                     std::is_same< InputType, typename Minus::D2 >::value &&
00196                     std::is_same< InputType, typename Minus::D3 >::value
00197                 ), "no_casting descriptor was set, but given minus operator has "
```

```
00198                 "incompatible domains with the given containers."
00199             );
00200             static_assert( !( descr & descriptors::no_casting ) || (
00201                     std::is_same< ResidualType, typename Divide::D1 >::value &&
00202                     std::is_same< ResidualType, typename Divide::D2 >::value &&
00203                     std::is_same< ResidualType, typename Divide::D3 >::value
00204                 ), "no_casting descriptor was set, but given divide operator has "
00205                 "incompatible domains with the given tolerance type."
00206             );
00207             static_assert( std::is_floating_point< ResidualType >::value,
00208                 "Require floating-point residual type."
00209             );
00210
00211             constexpr const Descriptor descr_dense = descr | descriptors::dense;
00212             const ResidualType zero_residual = ring.template getZero< ResidualType >();
00213             const IOType zero = ring.template getZero< IOType >();
00214             const size_t n = grb::ncols( A );
00215
00216             // dynamic checks
00217             {
00218                 const size_t m = grb::nrows( A );
00219                 if( size( x ) != n ) {
00220                     return MISMATCH;
00221                 }
00222                 if( size( b ) != m ) {
00223                     return MISMATCH;
00224                 }
00225                 if( size( r ) != n || size( u ) != n || size( temp ) != n ) {
00226                     std::cerr « "Error: provided workspace vectors are not of the correct "
00227                         « "length.\n";
00228                     return MISMATCH;
00229                 }
00230                 if( m != n ) {
00231                     std::cerr « "Warning: grb::algorithms::conjugate_gradient requires "
00232                         « "square input matrices, but a non-square input matrix was "
00233                         « "given instead.\n";
00234                     return ILLEGAL;
00235                 }
00236
00237                 // capacities
00238                 if( capacity( x ) != n ) {
00239                     return ILLEGAL;
00240                 }
00241                 if( capacity( r ) != n || capacity( u ) != n || capacity( temp ) != n ) {
00242                     return ILLEGAL;
00243                 }
00244
00245                 // others
00246                 if( tol <= zero_residual ) {
00247                     std::cerr « "Error: tolerance input to CG must be strictly positive\n";
00248                     return ILLEGAL;
00249                 }
00250             }
00251
00252             // set pure output fields to neutral defaults
00253             iterations = 0;
00254             residual = std::numeric_limits< double >::infinity();
00255
00256             // trivial shortcuts
00257             if( max_iterations == 0 ) {
00258                 return FAILED;
00259             }
00260
00261             // make x and b structurally dense (if not already) so that the remainder
00262             // algorithm can safely use the dense descriptor for faster operations
00263             {
00264                 RC rc = SUCCESS;
00265                 if( nnz( x ) != n ) {
00266                     rc = set< descriptors::invert_mask | descriptors::structural >(
00267                         x, x, zero
00268                     );
00269                 }
00270                 if( rc != SUCCESS ) {
00271                     return rc;
00272                 }
00273                 assert( nnz( x ) == n );
00274             }
00275
00276             IOType sigma, bnorm, alpha, beta;
00277
00278             // temp = 0
00279             grb::RC ret = grb::set( temp, 0 );
00280             assert( ret == SUCCESS );
00281
00282             // temp = A * x
00283             ret = ret ? ret : grb::mxv< descr_dense >( temp, A, x, ring );
00284             assert( ret == SUCCESS );
```

```
00285
00286             // r = b - temp;
00287             ret = ret ? ret : grb::set( r, zero );
00288             ret = ret ? ret : grb::foldl( r, b, ring.getAdditiveMonoid() );
00289             assert( nnz( r ) == n );
00290             assert( nnz( temp ) == n );
00291             ret = ret ? ret : grb::foldl< descr_dense >( r, temp, minus );
00292             assert( ret == SUCCESS );
00293             assert( nnz( r ) == n );
00294
00295             // u = r;
00296             ret = ret ? ret : grb::set( u, r );
00297             assert( ret == SUCCESS );
00298
00299             // sigma = r' * r;
00300             sigma = zero;
00301             if( grb::utils::is_complex< IOType >::value ) {
00302                 ret = ret ? ret : grb::eWiseLambda( [&temp,&r]( const size_t i ) {
00303                         temp[ i ] = grb::utils::is_complex< IOType >::conjugate( r[ i ] );
00304                     }, temp
00305                 );
00306                 ret = ret ? ret : grb::dot< descr_dense >( sigma, temp, r, ring );
00307             } else {
00308                 ret = ret ? ret : grb::dot< descr_dense >( sigma, r, r, ring );
00309             }
00310
00311             assert( ret == SUCCESS );
00312
00313             // bnorm = b' * b;
00314             bnorm = zero;
00315             if( grb::utils::is_complex< IOType >::value ) {
00316                 ret = ret ? ret : grb::eWiseLambda( [&temp,&b]( const size_t i ) {
00317                         temp[ i ] = grb::utils::is_complex< IOType >::conjugate( b[ i ] );
00318                     }, temp
00319                 );
00320                 ret = ret ? ret : grb::dot< descr_dense >( bnorm, temp, b, ring );
00321             } else {
00322                 ret = ret ? ret : grb::dot< descr_dense >( bnorm, b, b, ring );
00323             }
00324             assert( ret == SUCCESS );
00325
00326             if( ret == SUCCESS ) {
00327                 tol *= sqrt( grb::utils::is_complex< IOType >::modulus( bnorm ) );
00328             }
00329
00330             size_t iter = 0;
00331
00332             do {
00333                 assert( iter < max_iterations );
00334                 (void) ++iter;
00335
00336                 // temp = 0
00337                 ret = ret ? ret : grb::set( temp, 0 );
00338                 assert( ret == SUCCESS );
00339
00340                 // temp = A * u;
00341                 ret = ret ? ret : grb::mxv< descr_dense >( temp, A, u, ring );
00342                 assert( ret == SUCCESS );
00343
00344                 // beta = u' * temp
00345                 beta = zero;
00346                 if( grb::utils::is_complex< IOType >::value ) {
00347                     ret = ret ? ret : grb::eWiseLambda( [&u]( const size_t i ) {
00348                             u[ i ] = grb::utils::is_complex< IOType >::conjugate( u[ i ] );
00349                         }, u
00350                     );
00351                 }
00352                 ret = ret ? ret : grb::dot< descr_dense >( beta, temp, u, ring );
00353                 if( grb::utils::is_complex< IOType >::value ) {
00354                     ret = ret ? ret : grb::eWiseLambda( [&u]( const size_t i ) {
00355                             u[ i ] = grb::utils::is_complex< IOType >::conjugate( u[ i ] );
00356                         }, u
00357                     );
00358                 }
00359                 assert( ret == SUCCESS );
00360
00361                 // alpha = sigma / beta;
00362                 ret = ret ? ret : grb::apply( alpha, sigma, beta, divide );
00363                 assert( ret == SUCCESS );
00364
00365                 // x = x + alpha * u;
00366                 ret = ret ? ret : grb::eWiseMul< descr_dense >( x, alpha, u, ring );
00367                 assert( ret == SUCCESS );
00368
00369                 // temp = alpha .* temp
00370                 // Warning: operator-based foldr requires temp be dense
00371                 ret = ret ? ret : grb::foldr( alpha, temp, ring.getMultiplicativeMonoid() );
```

```
00372                 assert( ret == SUCCESS );
00373
00374                 // r = r - temp;
00375                 ret = ret ? ret : grb::foldl< descr_dense >( r, temp, minus );
00376                 assert( ret == SUCCESS );
00377
00378                 // beta = r' * r;
00379                 beta = zero;
00380                 if( grb::utils::is_complex< IOType >::value ) {
00381                     ret = ret ? ret : grb::eWiseLambda( [&temp,&r]( const size_t i ) {
00382                             temp[ i ] = grb::utils::is_complex< IOType >::conjugate( r[ i ] );
00383                         }, temp
00384                     );
00385                     ret = ret ? ret : grb::dot< descr_dense >( beta, temp, r, ring );
00386                 } else {
00387                     ret = ret ? ret : grb::dot< descr_dense >( beta, r, r, ring );
00388                 }
00389                 residual = grb::utils::is_complex< IOType >::modulus( beta );
00390                 assert( ret == SUCCESS );
00391
00392                 if( ret == SUCCESS ) {
00393                     if( sqrt( residual ) < tol || iter >= max_iterations ) {
00394                         break;
00395                     }
00396                 }
00397
00398                 // alpha = beta / sigma;
00399                 ret = ret ? ret : grb::apply( alpha, beta, sigma, divide );
00400                 assert( ret == SUCCESS );
00401
00402                 // temp = r + alpha * u;
00403                 ret = ret ? ret : grb::set( temp, r );
00404                 assert( ret == SUCCESS );
00405                 ret = ret ? ret : grb::eWiseMul< descr_dense >( temp, alpha, u, ring );
00406                 assert( ret == SUCCESS );
00407                 assert( nnz( temp ) == size( temp ) );
00408
00409                 // u = temp
00410                 std::swap( u, temp );
00411
00412                 sigma = beta;
00413             } while( ret == SUCCESS );
00414
00415             // output that is independent of error code
00416             iterations = iter;
00417
00418             // return correct error code
00419             if( ret == SUCCESS ) {
00420                 if( sqrt( residual ) >= tol ) {
00421                     // did not converge within iterations
00422                     return FAILED;
00423                 }
00424             }
00425             return ret;
00426         }
00427
00428     } // namespace algorithms
00429
00430 } // end namespace grb
00431
00432 #endif // end _H_GRB_ALGORITHMS_CONJUGATE_GRADIENT
00433
```

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.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_COSSIM
00028 #define _H_GRB_COSSIM
00029
00030 #include <graphblas.hpp>
00031 #include <graphblas/algorithms/norm.hpp>
00032
00033
00034 #define NO_CAST_ASSERT( x, y, z )
      \
00035     static_assert( x,
      \
00036         "\n\n"
      \
00037
      "*****************************************************************************************\n" \
00038         "*     ERROR      | " y " " z ".\n"
      \
00039
      "*****************************************************************************************\n" \
00040         "* Possible fix 1 | Remove no_casting from the template parameters in this call to " y ".
      \n" \
00041         "* Possible fix 2 | For all mismatches in the domains of input and output parameters w.r.t.
      \n" \
00042         "*                  the semiring domains, as specified in the documentation of the
      function.\n" \
00043         "*                  supply an input argument of the expected type instead.\n"
      \
```

```
00044         "* Possible fix 3 | Provide a compatible semiring where all domains match those of the
      input\n" \
00045         "*                   parameters, as specified in the documentation of the function.\n"
      \
00046
      "*********************************************************************************************\n" );
00047
00048
00049 namespace grb {
00050
00051     namespace algorithms {
00052
00104         template< Descriptor descr = descriptors::no_operation,
00105             typename OutputType,
00106             typename InputType1,
00107             typename InputType2,
00108             class Ring,
00109             class Division = grb::operators::divide<
00110                 typename Ring::D3, typename Ring::D3, typename Ring::D4
00111             >
00112         >
00113         RC cosine_similarity(
00114             OutputType &similarity,
00115             const Vector< InputType1 > &x, const Vector< InputType2 > &y,
00116             const Ring &ring = Ring(), const Division &div = Division()
00117         ) {
00118             static_assert( std::is_floating_point< OutputType >::value,
00119                 "Cosine similarity requires a floating-point output type." );
00120
00121             // static sanity checks
00122             NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) ||
00123                     std::is_same< InputType1, typename Ring::D1 >::value
00124                 ), "grb::algorithms::cosine_similarity",
00125                 "called with a left-hand vector value type that does not match the "
00126                 "first domain of the given semiring" );
00127             NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) ||
00128                     std::is_same< InputType2, typename Ring::D2 >::value
00129                 ), "grb::algorithms::cosine_similarity",
00130                 "called with a right-hand vector value type that does not match "
00131                 "the second domain of the given semiring" );
00132             NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) ||
00133                     std::is_same< OutputType, typename Ring::D4 >::value
00134                 ), "grb::algorithms::cosine_similarity",
00135                 "called with an output vector value type that does not match the "
00136                 "fourth domain of the given semiring" );
00137             NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) ||
00138                     std::is_same< typename Ring::D3, typename Ring::D4 >::value
00139                 ), "grb::algorithms::cosine_similarity",
00140                 "called with a semiring that has unequal additive input domains" );
00141
00142             const size_t n = size( x );
00143
00144             // run-time sanity checks
00145             if( n != size( y ) ) {
00146                 return MISMATCH;
00147             }
00148
00149             // check whether inputs are dense
00150             const bool dense = nnz( x ) == n && nnz( y ) == n;
00151
00152             // set return code
00153             RC rc = SUCCESS;
00154
00155             // compute-- choose method depending on we can stream once or need to stream
00156             // multiple times
00157             OutputType nominator, denominator;
00158             nominator = denominator = ring.template getZero< OutputType >();
00159             if( dense && grb::Properties<>::writableCaptured ) {
00160                 // lambda works, so we can stream each vector precisely once:
00161                 OutputType norm1, norm2;
00162                 norm1 = norm2 = ring.template getZero< OutputType >();
00163                 rc = grb::eWiseLambda(
00164                     [ &x, &y, &nominator, &norm1, &norm2, &ring ]( const size_t i ) {
00165                         const auto &mul = ring.getMultiplicativeOperator();
00166                         const auto &add = ring.getAdditiveOperator();
00167                         OutputType temp;
00168                         (void)grb::apply( temp, x[ i ], y[ i ], mul );
00169                         (void)grb::foldl( nominator, temp, add );
00170                         (void)grb::apply( temp, x[ i ], x[ i ], mul );
00171                         (void)grb::foldl( norm1, temp, add );
00172                         (void)grb::apply( temp, y[ i ], y[ i ], mul );
00173                         (void)grb::foldl( norm2, temp, add );
00174                     },
00175                     x, y );
00176                 denominator = sqrt( norm1 ) * sqrt( norm2 );
00177             } else {
00178                 // cannot stream each vector once, stream each one twice instead using
```

```
00179                 // standard grb functions
00180                 rc = grb::norm2( nominator, x, ring );
00181                 if( rc == SUCCESS ) {
00182                     rc = grb::norm2( denominator, y, ring );
00183                 }
00184                 if( rc == SUCCESS ) {
00185                     rc = grb::foldl( denominator, nominator,
00186                         ring.getMultiplicativeOperator() );
00187                 }
00188                 if( rc == SUCCESS ) {
00189                     rc = grb::dot( nominator, x, y, ring );
00190                 }
00191             }
00192
00193             // accumulate
00194             if( rc == SUCCESS ) {
00195                 // catch zeroes
00196                 if( denominator == ring.template getZero() ) {
00197                     return ILLEGAL;
00198                 }
00199                 if( nominator == ring.template getZero() ) {
00200                     return ILLEGAL;
00201                 }
00202                 rc = grb::apply( similarity, nominator, denominator, div );
00203             }
00204
00205             // done
00206             return rc;
00207         }
00208
00209     } // end namespace algorithms
00210
00211 } // end namespace grb
00212
00213 #undef NO_CAST_ASSERT
00214
00215 #endif // end _H_GRB_COSSIM
00216
```

10.9 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

10.9.1 Detailed Description

Implements k-means.

The state of the algorithms defined within are *experimental*.

Author

Verner Vlacic

10.10 kmeans.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_KMEANS
00028 #define _H_GRB_KMEANS
00029
00030 #include <chrono>
00031 #include <random>
00032
00033 #include <assert.h>
00034
00035 #include <graphblas.hpp>
00036
00037
00038 namespace grb {
00039
00040     namespace algorithms {
00041
00055         template<
00056             Descriptor descr = descriptors::no_operation,
00057             typename IOType = double,
00058             class Operator = operators::square_diff< IOType, IOType, IOType >
00059         >
00060         RC kpp_initialisation(
00061             Matrix< IOType > &K,
00062             const Matrix< IOType > &X,
00063             const Operator &dist_op = Operator()
00064         ) {
00065             // declare monoids and semirings
00066             Monoid< grb::operators::add< IOType >, grb::identities::zero > add_monoid;
00067             Monoid<
00068                 grb::operators::min< IOType >,
00069                 grb::identities::infinity
00070             > min_monoid;
00071             Semiring<
00072                 grb::operators::add< IOType >,
00073                 grb::operators::right_assign_if< bool, IOType, IOType >,
00074                 grb::identities::zero, grb::identities::logical_true
00075             > pattern_sum;
00076
00077             // runtime sanity checks: the row dimension of X should match the column
00078             // dimension of K
00079             if( ncols( K ) != nrows( X ) ) {
00080                 return MISMATCH;
00081             }
00082
00083             // running error code
00084             RC ret = SUCCESS;
00085
```

```
00086             // get problem dimensions
00087             const size_t n = ncols( X );
00088             const size_t m = nrows( X );
00089             const size_t k = nrows( K );
00090
00091             // declare vector of indices of columns of X selected as the initial
00092             // centroids
00093             Vector< size_t > selected_indices( k );
00094
00095             // declare column selection vector
00096             Vector< bool > col_select( n );
00097
00098             // declare selected point
00099             Vector< IOType > selected( m );
00100
00101             // declare vector of distances from the selected point
00102             Vector< IOType > selected_distances( n );
00103
00104             // declare vector of minimum distances to all points selected so far
00105             Vector< IOType > min_distances( n );
00106             ret = ret ? ret : grb::set( min_distances,
00107                 grb::identities::infinity< IOType >::value()
00108             );
00109
00110             // generate first centroid by selecting a column of X uniformly at random
00111
00112             size_t i;
00113             {
00114                 const size_t seed_uniform =
00115                     std::chrono::system_clock::now().time_since_epoch().count();
00116                 std::default_random_engine random_generator( seed_uniform );
00117                 std::uniform_int_distribution< size_t > uniform( 0, n - 1 );
00118                 i = uniform( random_generator );
00119             }
00120
00121             for( size_t l = 0; ret == SUCCESS && l < k; ++l ) {
00122
00123                 ret = grb::clear( col_select );
00124                 ret = ret ? ret : grb::clear( selected );
00125                 ret = ret ? ret : grb::clear( selected_distances );
00126
00127                 ret = ret ? ret : grb::setElement( selected_indices, i, l );
00128
00129                 ret = ret ? ret : grb::setElement( col_select, true, i );
00130
00131                 ret = ret ? ret : grb::vxm< grb::descriptors::transpose_matrix >(
00132                     selected, col_select, X, pattern_sum );
00133
00134                 ret = ret ? ret : grb::vxm( selected_distances, selected, X, add_monoid,
00135                     dist_op );
00136
00137                 ret = ret ? ret : grb::foldl( min_distances, selected_distances,
00138                     min_monoid );
00139
00140                 // TODO the remaining part of the loop should be replaced with the alias
00141                 //      algorithm
00142
00143                 IOType range = add_monoid.template getIdentity< IOType >();
00144                 ret = ret ? ret : grb::foldl( range, min_distances, add_monoid );
00145
00146                 double sample = -1;
00147                 if( ret == SUCCESS ) {
00148                     {
00149                         const size_t seed =
00150                             std::chrono::system_clock::now().time_since_epoch().count();
00151                         std::default_random_engine generator( seed );
00152                         std::uniform_real_distribution< double > uniform( 0, 1 );
00153                         sample = uniform( generator );
00154                     }
00155                     ret = grb::collectives<>::broadcast( sample, 0 );
00156                 }
00157                 assert( sample >= 0 );
00158
00159                 // The following is not standard ALP/GraphBLAS and does not work for P>1
00160                 //    (TODO internal issue #320)
00161                 if( ret == SUCCESS ) {
00162                     assert( grb::spmd<>::nprocs() == 1 );
00163                     IOType * const raw = internal::getRaw( selected_distances );
00164                     IOType running_sum = 0;
00165                     i = 0;
00166                     do {
00167                         running_sum += static_cast< double >( raw[ i ] ) / range;
00168                     } while( running_sum < sample && ++i < n );
00169                     i = ( i == n ) ? n - 1 : i;
00170                 }
00171             }
00172
```

```
00173             // create the matrix K by selecting the columns of X indexed by
00174             // selected_indices
00175
00176             // declare pattern matrix
00177             Matrix< void > M( k, n );
00178             ret = ret ? ret : grb::resize( M, n );
00179
00180             if( ret == SUCCESS ) {
00181                 auto converter = grb::utils::makeVectorToMatrixConverter< void, size_t >(
00182                     selected_indices, [](const size_t &ind, const size_t &val ) {
00183                         return std::make_pair( ind, val );
00184                     }
00185                 );
00186                 ret = grb::buildMatrixUnique( M, converter.begin(), converter.end(),
00187                     PARALLEL );
00188             }
00189
00190             ret = ret ? ret : grb::mxm< descriptors::transpose_right >( K, M, X,
00191                 pattern_sum, RESIZE );
00192             ret = ret ? ret : grb::mxm< descriptors::transpose_right >( K, M, X,
00193                 pattern_sum );
00194
00195             if( ret != SUCCESS ) {
00196                 std::cout « "\tkpp finished with unexpected return code!" « std::endl;
00197             }
00198
00199             return ret;
00200         }
00201
00219         template<
00220             Descriptor descr = descriptors::no_operation,
00221             typename IOType = double,
00222             class Operator = operators::square_diff< IOType, IOType, IOType >
00223         >
00224         RC kmeans_iteration(
00225             Matrix< IOType > &K,
00226             Vector< std::pair< size_t, IOType > > &clusters_and_distances,
00227             const Matrix< IOType > &X,
00228             const size_t max_iter = 1000,
00229             const Operator &dist_op = Operator()
00230         ) {
00231             // declare monoids and semirings
00232
00233             typedef std::pair< size_t, IOType > indexIOType;
00234
00235             Monoid< grb::operators::add< IOType >, grb::identities::zero > add_monoid;
00236
00237             Monoid<
00238                 grb::operators::argmin< size_t, IOType >,
00239                 grb::identities::infinity
00240             > argmin_monoid;
00241
00242             Monoid<
00243                 grb::operators::logical_and< bool >,
00244                 grb::identities::logical_true
00245             > comparison_monoid;
00246
00247             Semiring<
00248                 grb::operators::add< IOType >,
00249                 grb::operators::right_assign_if< bool, IOType, IOType >,
00250                 grb::identities::zero, grb::identities::logical_true
00251             > pattern_sum;
00252
00253             Semiring<
00254                 grb::operators::add< size_t >,
00255                 grb::operators::right_assign_if< size_t, size_t, size_t >,
00256                 grb::identities::zero, grb::identities::logical_true
00257             > pattern_count;
00258
00259             // runtime sanity checks: the row dimension of X should match the column
00260             // dimension of K
00261             if( ncols( K ) != nrows( X ) ) {
00262                 return MISMATCH;
00263             }
00264             if( size( clusters_and_distances ) != ncols( X ) ) {
00265                 return MISMATCH;
00266             }
00267
00268             // running error code
00269             RC ret = SUCCESS;
00270
00271             // get problem dimensions
00272             const size_t n = ncols( X );
00273             const size_t m = nrows( X );
00274             const size_t k = nrows( K );
00275
00276             // declare distance matrix
```

```
00277             Matrix< IOType > Dist( k, n );
00278
00279             // declare and initialise labels vector and ones vectors
00280             Vector< size_t > labels( k );
00281             Vector< bool > n_ones( n ), m_ones( m );
00282
00283             ret = ret ? ret : grb::set< grb::descriptors::use_index >( labels, 0 );
00284             ret = ret ? ret : grb::set( n_ones, true );
00285             ret = ret ? ret : grb::set( m_ones, true );
00286
00287             // declare pattern matrix
00288             Matrix< void > M( k, n );
00289             ret = ret ? ret : grb::resize( M, n );
00290
00291             // declare the sizes vector
00292             Vector< size_t > sizes( k );
00293
00294             // declare auxiliary vectors and matrices
00295             Matrix< IOType > K_aux( k, m );
00296             Matrix< size_t > V_aux( k, m );
00297
00298             // control variables
00299             size_t iter = 0;
00300             Vector< indexIOType > clusters_and_distances_prev( n );
00301             bool converged;
00302
00303             do {
00304                 (void) ++iter;
00305
00306                 ret = ret ? ret : grb::set( clusters_and_distances_prev,
00307                     clusters_and_distances );
00308
00309                 ret = ret ? ret : mxm( Dist, K, X, add_monoid, dist_op, RESIZE );
00310                 ret = ret ? ret : mxm( Dist, K, X, add_monoid, dist_op );
00311
00312                 ret = ret ? ret : vxm( clusters_and_distances, labels, Dist, argmin_monoid,
00313                     operators::zip< size_t, IOType >() );
00314
00315                 auto converter = grb::utils::makeVectorToMatrixConverter<
00316                     void, indexIOType
00317                 > (
00318                     clusters_and_distances,
00319                     []( const size_t &ind, const indexIOType &pair ) {
00320                         return std::make_pair( pair.first, ind );
00321                     }
00322                 );
00323
00324                 ret = ret ? ret : grb::buildMatrixUnique( M, converter.begin(),
00325                     converter.end(), PARALLEL );
00326
00327                 ret = ret ? ret : grb::mxm< descriptors::transpose_right >( K_aux, M, X,
00328                     pattern_sum, RESIZE );
00329                 ret = ret ? ret : grb::mxm< descriptors::transpose_right >( K_aux, M, X,
00330                     pattern_sum );
00331
00332                 ret = ret ? ret : grb::mxv( sizes, M, n_ones, pattern_count );
00333
00334                 ret = ret ? ret : grb::outer( V_aux, sizes, m_ones,
00335                     operators::left_assign_if< IOType, bool, IOType >(), RESIZE );
00336                 ret = ret ? ret : grb::outer( V_aux, sizes, m_ones,
00337                     operators::left_assign_if< IOType, bool, IOType >() );
00338
00339                 ret = ret ? ret : eWiseApply( K, V_aux, K_aux,
00340                     operators::divide_reverse< size_t, IOType, IOType >(), RESIZE );
00341                 ret = ret ? ret : eWiseApply( K, V_aux, K_aux,
00342                     operators::divide_reverse< size_t, IOType, IOType >() );
00343
00344                 converged = true;
00345                 ret = ret ? ret : grb::dot(
00346                     converged,
00347                     clusters_and_distances_prev, clusters_and_distances,
00348                     comparison_monoid,
00349                     grb::operators::equal_first< indexIOType, indexIOType, bool >()
00350                 );
00351
00352             } while( ret == SUCCESS && !converged && iter < max_iter );
00353
00354             if( iter == max_iter ) {
00355                 std::cout « "\tkmeans reached maximum number of iterations!" « std::endl;
00356                 return FAILED;
00357             }
00358             if( converged ) {
00359                 std::cout « "\tkmeans converged successfully after " « iter
00360                     « " iterations." « std::endl;
00361                 return SUCCESS;
00362             }
00363
```

```
00364             std::cout « "\tkmeans finished with unexpected return code!" « std::endl;
00365             return ret;
00366         }
00367
00368     } // namespace algorithms
00369
00370 } // namespace grb
00371
00372 #endif // end _H_GRB_KMEANS
00373
```

10.11 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.

10.11.1 Detailed Description

Implements the k-hop nearest neighbours from a given source vertex.

Author

A. N. Yzelman

Date

: 27th of April, 2017

10.12 knn.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_KNN
00028 #define _H_GRB_KNN
00029
00030 #include "graphblas/algorithms/mpv.hpp"
00031
00032 #include <graphblas.hpp>
00033
00034
00035 namespace grb {
00036
00037     namespace algorithms {
00038
00081         template< Descriptor descr, typename OutputType, typename InputType >
00082         RC knn(
00083             Vector< OutputType > &u, const Matrix< InputType > &A,
00084             const size_t source, const size_t k,
00085             Vector< bool > &buf1
00086         ) {
00087             // the nearest-neighbourhood ring
00088             Semiring<
00089                 operators::logical_or< bool >, operators::logical_and< bool >,
00090                 identities::logical_false, identities::logical_true
00091             > ring;
00092
00093             // check input
00094             const size_t n = nrows( A );
00095             if( n != ncols( A ) ) {
00096                 return MISMATCH;
00097             }
00098             if( size( buf1 ) != n ) {
00099                 return MISMATCH;
00100             }
00101             if( size( u ) != n ) {
00102                 return MISMATCH;
00103             }
00104             if( capacity( u ) != n ) {
00105                 return ILLEGAL;
00106             }
00107             if( capacity( buf1 ) != n ) {
00108                 return ILLEGAL;
00109             }
00110
00111             // prepare
00112             RC ret = SUCCESS;
00113             if( nnz( u ) != 0 ) {
00114                 ret = clear( u );
00115             }
00116             if( nnz( buf1 ) != 0 ) {
00117                 ret = ret ? ret : clear( buf1 );
00118             }
00119 #ifdef _DEBUG
00120             std::cout « "grb::algorithms::knn called with source " « source « " "
00121                 « "and k " « k « ".\n";
00122 #endif
00123             ret = ret ? ret : setElement( buf1, true, source );
00124
00125             // do sparse matrix powers on the given ring
00126             if( ret == SUCCESS ) {
00127                 if( descr & descriptors::transpose_matrix ) {
00128                     ret = mpv< (descr | descriptors::add_identity) &
00129                         ~( descriptors::transpose_matrix )
00130                     >( u, A, k, buf1, buf1, ring );
00131                 } else {
00132                     ret = mpv< descr | descriptors::add_identity |
00133                         descriptors::transpose_matrix
```

```
00134                     >( u, A, k, buf1, buf1, ring );
00135                 }
00136             }
00137
00138             // done
00139             return ret;
00140         }
00141
00142     } // namespace algorithms
00143
00144 } // namespace grb
00145
00146 #endif
00147
```

10.13 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.

10.13.1 Detailed Description

Implements label propagation.

Author

J. M. Nash

Date

21st of March, 2017

10.14 label.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_LABEL
00028 #define _H_GRB_LABEL
00029
00030 #include <iostream>
00031
00032 #include <graphblas.hpp>
00033
00034
00035 namespace grb {
00036
00037     namespace algorithms {
00038
00039 #ifdef _DEBUG
00040         constexpr size_t MaxPrinting = 20;
00041         constexpr size_t MaxAnyPrinting = 100;
00042
00043         // take a vector and display with a message
00044         static void printVector(
00045             const Vector< double > &v, const std::string message
00046         ) {
00047             size_t zeros = 0;
00048             size_t ones = 0;
00049             size_t size = grb::size( v );
00050             if( size > MaxAnyPrinting ) {
00051                 return;
00052             }
00053             std::cerr « "\t " « message « ": ";
00054             for( Vector< double >::const_iterator it = v.begin(); it != v.end(); ++it ) {
00055                 const std::pair< size_t, double > iter = *it;
00056                 const double val = iter.second;
00057                 if( val < INFINITY ) {
00058                     if( size > MaxPrinting ) {
00059                         zeros += ( val == 0 ) ? 1 : 0;
00060                         ones += ( val == 1 ) ? 1 : 0;
00061                     } else {
00062                         std::cerr « val « " ";
00063                     }
00064                 }
00065             }
00066             if( size > MaxPrinting ) {
00067                 std::cerr « zeros « " zeros; " « ones « " ones.";
00068             }
00069             std::cerr « "\n";
00070         }
00071 #endif
00072
00122         template< typename IOType >
00123         RC label(
00124             Vector< IOType > &out,
00125             const Vector< IOType > &y, const Matrix< IOType > &W,
00126             const size_t n, const size_t l,
00127             const size_t maxIterations = 1000
00128         ) {
00129             // label propagation vectors and matrices operate over the real domain
00130             Semiring<
00131                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
00132                 grb::identities::zero, grb::identities::one
00133             > reals;
00134             grb::operators::not_equal< IOType, IOType, bool > notEqualOp;
00135             grb::Monoid<
00136                 grb::operators::logical_or< bool >,
00137                 grb::identities::logical_false
00138             > orMonoid;
00139             const IOType zero = reals.template getZero< IOType >();
00140
```

```
00141             // dynamic checks
00142             if( nrows( W ) != n || ncols( W ) != n ||
00143                 size( y ) != n || size( out ) != n
00144             ) {
00145                 return ILLEGAL;
00146             }
00147             if( capacity( out ) != n ) {
00148                 return ILLEGAL;
00149             }
00150             if( n == 0 ) {
00151                 return SUCCESS;
00152             }
00153             if( l == 0 ) {
00154                 return ILLEGAL;
00155             }
00156
00157             const size_t s = spmd<>::pid();
00158
00159             // compute the diagonal matrix D from the weight matrix W
00160             // we represent D as a vector so we can use it to generate the probabilities
00161             // matrix P
00162             Vector< IOType > multiplier( n );
00163             RC ret = set( multiplier, static_cast< IOType >(1) ); // a vector of 1's
00164
00165             Vector< IOType > diagonals( n );
00166             ret = ret ? ret : set( diagonals, zero );
00167             ret = ret ? ret : mxv< descriptors::dense >(
00168                 diagonals, W, multiplier, reals
00169             ); // W*multiplier will sum each row
00170 #ifdef _DEBUG
00171             printVector( diagonals, "diagonals matrix in vector form" );
00172 #endif
00173
00174             // compute the probabilistic transition matrix P as inverse of D * W
00175             // use diagonals vector to directly compute probabilistic transition matrix P
00176             // only the existing non-zero elements in W will map to P
00177             // the inverse of D is represented via the inverse element in the diagonals
00178             // vector
00179             //
00180             // update: the application of Dinv is now done within a lambda following the
00181             //         mxv on the original matrix P
00182
00183             // make diagonals equal its inverse
00184             ret = ret ? ret : eWiseLambda( [ &diagonals ]( const size_t i ) {
00185                     diagonals[ i ] = 1.0 / diagonals[ i ];
00186                 }, diagonals
00187             );
00188
00189             // set up current and new solution functions
00190             Vector< IOType > f( n );
00191             Vector< IOType > fNext( n );
00192             Vector< bool > mask( n );
00193             for( size_t i = 0; ret == SUCCESS && i < l; ++i ) {
00194                 ret = setElement( mask, true, i );
00195             }
00196
00197             // fix f = y for the input set of labels
00198             ret = ret ? ret : set( f, y );
00199
00200             // whether two successive solutions are different
00201             // initially set to true so that the computation may begin
00202             bool different = true;
00203             // compute f as P*f
00204             // main loop completes when function f is stable
00205             size_t iter = 1;
00206             while( ret == SUCCESS && different && iter < maxIterations ) {
00207
00208 #ifdef _DEBUG
00209                 if( n < MaxAnyPrinting ) {
00210                     std::cerr « "\t iteration " « iter « "\n";
00211                 }
00212
00213                 // fNext = P*f
00214                 std::cerr « "\t pre- set/mxv nnz( f ) = " « nnz( f ) « ", "
00215                     « "fNext = " « nnz( fNext ) « "\n";
00216 #endif
00217                 ret = ret ? ret : set( fNext, zero );
00218                 ret = ret ? ret : mxv( fNext, W, f, reals );
00219 #ifdef _DEBUG
00220                 std::cerr « "\t post-set/mxv nnz( f ) = " « nnz( f ) « ", "
00221                     « "nnz( fNext ) = " « nnz( fNext ) « "\n";
00222                 printVector( f, "Previous iteration solution" );
00223                 printVector( fNext, "New iteration solution" );
00224 #endif
00225
00226                 // maintain the solution function in domain {0,1}
00227                 // can use the masked variant of vector assign when available ?
```

```
00228                 ret = ret ? ret : eWiseLambda( [ &fNext, &diagonals ]( const size_t i ) {
00229                         fNext[ i ] = ( fNext[ i ] * diagonals[ i ] < 0.5 ? 0 : 1 );
00230                     }, diagonals, fNext
00231                 );
00232 #ifdef _DEBUG
00233                 printVector( fNext, "New iteration solution after threshold cutoff" );
00234                 std::cerr « "\t pre-set  nnz( fNext ) = " « nnz( fNext ) « ", "
00235                     « "nnz( mask ) = " « nnz( mask ) « "\n";
00236 #endif
00237                 // clamps the first l labelled nodes
00238                 ret = ret ? ret : foldl(
00239                     fNext, mask,
00240                     f,
00241                     grb::operators::right_assign< IOType >()
00242                 );
00243                 assert( ret == SUCCESS );
00244 #ifdef _DEBUG
00245                 std::cerr « "\t post-set nnz( fNext ) = " « nnz( fNext ) « "\n";
00246                 printVector(
00247                     fNext,
00248                     "New iteration solution after threshold cutoff and clamping"
00249                 );
00250 #endif
00251                 // test for stability
00252                 different = false;
00253                 ret = ret ? ret : dot( different, f, fNext, orMonoid, notEqualOp );
00254
00255                 // update f for the next iteration
00256 #ifdef _DEBUG
00257                 std::cerr « "\t pre-set  nnz(f) = " « nnz( f ) « "\n";
00258 #endif
00259                 std::swap( f, fNext );
00260 #ifdef _DEBUG
00261                 std::cerr « "\t post-set nnz(f) = " « nnz( f ) « "\n";
00262 #endif
00263                 // go to next iteration
00264                 (void) ++iter;
00265             }
00266
00267             if( ret == SUCCESS ) {
00268                 if( different ) {
00269                     if( s == 0 ) {
00270                         std::cerr « "Info: label propagation did not converge after "
00271                             « (iter-1) « " iterations\n";
00272                     }
00273                     return FAILED;
00274                 } else {
00275                     if( s == 0 ) {
00276                         std::cerr « "Info: label propagation converged in "
00277                             « (iter-1) « " iterations\n";
00278                     }
00279                     std::swap( out, f );
00280                     return SUCCESS;
00281                 }
00282             }
00283
00284             // done
00285             if( s == 0 ) {
00286                 std::cerr « "Warning: label propagation exiting with " « toString(ret)
00287                     « "\n";
00288             }
00289             return ret;
00290         }
00291
00292     } // namespace algorithms
00293
00294 } // namespace grb
00295
00296 #endif // end _H_GRB_LABEL
00297
```

10.15 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.

10.15.1 Detailed Description

Implements the matrix powers kernel $y = A^k x$ over arbitrary semirings.

Author

A. N. Yzelman

Date

30th of March 2017

10.16 mpv.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_ALGORITHMS_MPV
00028 #define _H_GRB_ALGORITHMS_MPV
00029
00030 #include <graphblas.hpp>
00031
00032
00033 namespace grb {
00034
00035     namespace algorithms {
00036
00093         template< Descriptor descr, class Ring, typename IOType, typename InputType >
00094         RC mpv(
00095             Vector< IOType > &u,
00096             const Matrix< InputType > &A, const size_t k,
00097             const Vector< IOType > &v,
00098             Vector< IOType > &temp,
00099             const Ring &ring
00100         ) {
00101             static_assert( !(descr & descriptors::no_casting) ||
00102                     (std::is_same< IOType, typename Ring::D4 >::value &&
00103                         std::is_same< InputType, typename Ring::D2 >::value &&
```

```
00104                         std::is_same< IOType, typename Ring::D1 >::value &&
00105                         std::is_same< IOType, typename Ring::D3 >::value
00106                     ),
00107                 "grb::mpv : some containers were passed with element types that do not"
00108                 "match the given semiring domains."
00109             );
00110
00111             // runtime check
00112             const size_t n = nrows( A );
00113             if( n != ncols( A ) ) {
00114                 return ILLEGAL;
00115             }
00116             if( size( u ) != n || n != size( v ) ) {
00117                 return MISMATCH;
00118             }
00119             if( size( temp ) != n ) {
00120                 return MISMATCH;
00121             }
00122             if( capacity( u ) != n ) {
00123                 return ILLEGAL;
00124             }
00125             if( capacity( temp ) != n ) {
00126                 return ILLEGAL;
00127             }
00128             // catch trivial case
00129             if( k == 0 ) {
00130                 return set< descr >( u, v );
00131             }
00132             // otherwise, do at least one multiplication
00133 #ifdef _DEBUG
00134             std::cout « "init: input vector nonzeroes is " « grb::nnz( v ) « ".\n";
00135 #endif
00136             RC ret = mxv< descr >( u, A, v, ring );
00137             if( k == 1 ) {
00138                 return ret;
00139             }
00140             // do any remaining multiplications using a temporary output vector
00141             bool copy;
00142             ret = ret ? ret : clear( temp );
00143             for( size_t iterate = 1; ret == SUCCESS && iterate < k; iterate += 2 ) {
00144                 // multiply with output into temporary
00145                 copy = true;
00146 #ifdef _DEBUG
00147                 std::cout « "up: input vector nonzeroes is " « grb::nnz( u ) « "\n";
00148 #endif
00149                 ret = mxv< descr >( temp, A, u, ring );
00150                 // check if this was the final multiplication
00151                 assert( iterate <= k );
00152                 if( iterate + 1 == k || ret != SUCCESS ) {
00153                     break;
00154                 }
00155                 // multiply with output into u
00156                 copy = false;
00157 #ifdef _DEBUG
00158                 std::cout « "down: input vector nonzeroes is " « grb::nnz( temp ) « "\n";
00159 #endif
00160                 ret = mxv< descr >( u, A, temp, ring );
00161             }
00162
00163             // swap u and temp, if required
00164             if( ret == SUCCESS && copy ) {
00165                 std::swap( u, temp );
00166             }
00167
00168             // done
00169             return ret;
00170         }
00171
00172     } // namespace algorithms
00173
00174 } // namespace grb
00175
00176 #endif // end "_H_GRB_ALGORITHMS_MPV"
00177
```

10.17 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.

10.17.1 Detailed Description

Implements the 2-norm.

Author

A. N. Yzelman

Date

17th of March 2022

10.18 norm.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00034 #ifndef _H_GRB_ALGORITHMS_NORM
00035 #define _H_GRB_ALGORITHMS_NORM
00036
00037 #include <graphblas.hpp>
00038
00039 #include <cmath> // for std::sqrt
00040
00041
00042 namespace grb {
00043
00044     namespace algorithms {
00045
00071         template<
00072             Descriptor descr = descriptors::no_operation, class Ring,
00073             typename InputType, typename OutputType,
```

```
00074             Backend backend, typename Coords
00075         >
00076         RC norm2( OutputType &x,
00077             const Vector< InputType, backend, Coords > &y,
00078             const Ring &ring = Ring(),
00079             const typename std::enable_if<
00080                 std::is_floating_point< OutputType >::value,
00081             void >::type * const = nullptr
00082         ) {
00083             RC ret = grb::dot< descr >( x, y, y, ring );
00084             if( ret == SUCCESS ) {
00085                 x = sqrt( x );
00086             }
00087             return ret;
00088         }
00089
00090     }
00091 }
00092
00093 #endif // end "_H_GRB_ALGORITHMS_NORM"
00094
```

10.19 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.

10.19.1 Detailed Description

Implements the (strongly) connected components algorithm over undirected graphs using the ALP/Pregel interface.

Author

: A. N. Yzelman.

10.20 pregel_connected_components.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_PREGEL_CONNECTEDCOMPONENTS
00028 #define _H_GRB_PREGEL_CONNECTEDCOMPONENTS
00029
00030 #include <graphblas/interfaces/pregel.hpp>
00031
00032
00033 namespace grb {
00034
00035     namespace algorithms {
00036
00037         namespace pregel {
00038
00047             template< typename VertexIDType >
00048             struct ConnectedComponents {
00049
00054                 struct Data {};
00055
00081                 static void program(
00082                     VertexIDType &current_max_ID,
00083                     const VertexIDType &incoming_message,
00084                     VertexIDType &outgoing_message,
00085                     const Data &parameters,
00086                     grb::interfaces::PregelState &pregel
00087                 ) {
00088                     (void) parameters;
00089                     if( pregel.round > 0 ) {
00090                         if( pregel.indegree == 0 ) {
00091                             pregel.voteToHalt = true;
00092                         } else if( current_max_ID < incoming_message ) {
00093                             current_max_ID = incoming_message;
00094                         } else {
00095                             pregel.voteToHalt = true;
00096                         }
00097                     }
00098                     if( pregel.outdegree > 0 ) {
00099                         outgoing_message = current_max_ID;
00100                     } else {
00101                         pregel.voteToHalt = true;
00102                     }
00103                 }
00104
00124                 template< typename PregelType >
00125                 static grb::RC execute(
00126                     grb::interfaces::Pregel< PregelType > &pregel,
00127                     grb::Vector< VertexIDType > &group_ids,
00128                     const size_t max_steps = 0,
00129                     size_t * const steps_taken = nullptr
00130                 ) {
00131                     const size_t n = pregel.num_vertices();
00132                     if( grb::size( group_ids ) != n ) {
00133                         return MISMATCH;
00134                     }
00135
00136                     grb::RC ret = grb::set< grb::descriptors::use_index >( group_ids, 1 );
00137                     if( ret != SUCCESS ) {
00138                         return ret;
00139                     }
00140
00141                     grb::Vector< VertexIDType > in( n );
00142                     grb::Vector< VertexIDType > out( n );
00143                     grb::Vector< VertexIDType > out_buffer = interfaces::config::out_sparsify
00144                         ? grb::Vector< VertexIDType >( n )
00145                         : grb::Vector< VertexIDType >( 0 );
00146
00147                     size_t steps;
```

```
00148
00149                     ret = pregel.template execute<
00150                         grb::operators::max< VertexIDType >,
00151                         grb::identities::negative_infinity
00152                     > (
00153                         program,
00154                         group_ids,
00155                         Data(),
00156                         in, out,
00157                         steps,
00158                         out_buffer,
00159                         max_steps
00160                     );
00161
00162                     if( ret == grb::SUCCESS && steps_taken != nullptr ) {
00163                         *steps_taken = steps;
00164                     }
00165
00166                     return ret;
00167                 }
00168
00169             };
00170
00171         } //end namespace 'grb::algorithms::pregel'
00172
00173     } // end namespace "grb::algorithms"
00174
00175 } // end namespace "grb"
00176
00177 #endif
00178
```

10.21 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.

10.21.1 Detailed Description

Implements a traditional vertex-centric page ranking algorithm using ALP/Pregel.

Author

A. N. Yzelman

10.22 pregel_pagerank.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_PREGEL_PAGERANK
00028 #define _H_GRB_PREGEL_PAGERANK
00029
00030 #include <graphblas/interfaces/pregel.hpp>
00031
00032
00033 namespace grb {
00034
00035     namespace algorithms {
00036
00037         namespace pregel {
00038
00053             template< typename IOType, bool localConverge >
00054             struct PageRank {
00055
00059                 struct Data {
00060
00064                     IOType alpha = 0.15;
00065
00069                     IOType tolerance = 0.00001;
00070
00071                 };
00072
00087                 static void program(
00088                     IOType &current_score,
00089                     const IOType &incoming_message,
00090                     IOType &outgoing_message,
00091                     const Data &parameters,
00092                     grb::interfaces::PregelState &pregel
00093                 ) {
00094                     // initialise
00095                     if( pregel.round == 0 ) {
00096                         current_score = static_cast< IOType >( 1 );
00097                     }
00098
00099 #ifdef _DEBUG
00100                     // when in debug mode, probably one does not wish to track the state of
00101                     // each vertex individually, hence we include a simple guard by default:
00102                     const bool dbg = pregel.vertexID == 0;
00103                     if( dbg ) {
00104                         std::cout « "ID: " « pregel.vertexID « "\n"
00105                             « "\t active: " « pregel.active « "\n"
00106                             « "\t round: " « pregel.round « "\n"
00107                             « "\t previous score: " « current_score « "\n"
00108                             « "\t incoming message: " « incoming_message « "\n";
00109                     }
00110 #endif
00111
00112                     // compute
00113                     if( pregel.round > 0 ) {
00114                         const IOType old_score = current_score;
00115                         current_score = parameters.alpha +
00116                             (static_cast< IOType >(1) - parameters.alpha) * incoming_message;
00117                         if( fabs(current_score-old_score) < parameters.tolerance ) {
00118 #ifdef _DEBUG
00119                             std::cout « "\t\t vertex " « pregel.vertexID « " converged\n";
00120 #endif
00121                             if( localConverge ) {
00122                                 pregel.active = false;
00123                             } else {
00124                                 pregel.voteToHalt = true;
00125                             }
00126                         }
00127                     }
00128
```

```
00129                     // broadcast
00130                     if( pregel.outdegree > 0 ) {
00131                         outgoing_message =
00132                             current_score /
00133                             static_cast< IOType >(pregel.outdegree);
00134                     }
00135
00136 #ifdef _DEBUG
00137                     if( dbg ) {
00138                         std::cout « "\t current score: " « current_score « "\n"
00139                             « "\t voteToHalt: " « pregel.voteToHalt « "\n"
00140                             « "\t outgoing message: " « outgoing_message « "\n";
00141                     }
00142 #endif
00143
00144                 }
00145
00182                 template< typename PregelType >
00183                 static grb::RC execute(
00184                     grb::interfaces::Pregel< PregelType > &pregel,
00185                     grb::Vector< IOType > &scores,
00186                     size_t &steps_taken,
00187                     const Data &parameters = Data(),
00188                     const size_t max_steps = 0
00189                 ) {
00190                     const size_t n = pregel.num_vertices();
00191                     if( grb::size( scores ) != n ) {
00192                         return MISMATCH;
00193                     }
00194
00195                     grb::Vector< IOType > in( n );
00196                     grb::Vector< IOType > out( n );
00197                     grb::Vector< IOType > out_buffer = interfaces::config::out_sparsify
00198                         ? grb::Vector< IOType >( n )
00199                         : grb::Vector< IOType >( 0 );
00200
00201                     return pregel.template execute<
00202                             grb::operators::add< IOType >,
00203                             grb::identities::zero
00204                         > (
00205                             program,
00206                             scores,
00207                             parameters,
00208                             in, out,
00209                             steps_taken,
00210                             out_buffer,
00211                             max_steps
00212                         );
00213                 }
00214
00215             };
00216
00217         } //end namespace 'grb::algorithms::pregel'
00218
00219     } // end namespace "grb::algorithms"
00220
00221 } // end namespace "grb"
00222
00223 #endif
00224
```

10.23 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.

10.23.1 Detailed Description

Implements the canonical PageRank algorithm by Brin and Page.

Author

A. N. Yzelman

Date

: 21st of March, 2017

10.24 simple_pagerank.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_PAGERANK
00028 #define _H_GRB_PAGERANK
00029
00030 #include <graphblas.hpp>
00031
00032 #ifndef _GRB_NO_STDIO
00033 #include <iostream>
00034 #endif
00035
00036
00037 namespace grb {
00038
00039     namespace algorithms {
00040
00130         template<
00131             Descriptor descr = descriptors::no_operation,
00132             typename IOType, typename NonzeroT
00133         >
00134         RC simple_pagerank(
00135             Vector< IOType > &pr,
00136             const Matrix< NonzeroT > &L,
00137             Vector< IOType > &pr_next,
00138             Vector< IOType > &pr_nextnext,
00139             Vector< IOType > &row_sum,
00140             const IOType alpha = 0.85,
00141             const IOType conv = 0.0000001,
00142             const size_t max = 1000,
00143             size_t * const iterations = nullptr,
```

```
00144             double * const quality = nullptr
00145         ) {
00146             grb::Monoid<
00147                 grb::operators::add< IOType >,
00148                 grb::identities::zero
00149             > addM;
00150             grb::Semiring<
00151                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
00152                 grb::identities::zero, grb::identities::one
00153             > realRing;
00154 #ifdef _DEBUG
00155             const auto s = spmd<>::pid();
00156 #endif
00157             const size_t n = nrows( L );
00158             const IOType zero = realRing.template getZero< IOType >();
00159
00160             // runtime sanity checks
00161             {
00162                 if( n != ncols( L ) ) {
00163                     return ILLEGAL;
00164                 }
00165                 if( size( pr ) != n ) {
00166                     return MISMATCH;
00167                 }
00168                 if( size( pr ) != n ) {
00169                     return MISMATCH;
00170                 }
00171                 if( size( pr_next ) != n ||
00172                     size( pr_nextnext ) != n ||
00173                     size( row_sum ) != n
00174                 ) {
00175                     return MISMATCH;
00176                 }
00177                 if( capacity( pr ) != n ) {
00178                     return ILLEGAL;
00179                 }
00180                 if( capacity( pr_next ) != n ||
00181                     capacity( pr_nextnext ) != n ||
00182                     capacity( row_sum ) != n
00183                 ) {
00184                     return ILLEGAL;
00185                 }
00186                 // alpha must be within 0 and 1 (both exclusive)
00187                 if( alpha <= 0 || alpha >= 1 ) {
00188                     return ILLEGAL;
00189                 }
00190                 // max must be larger than 0
00191                 if( max <= 0 ) {
00192                     return ILLEGAL;
00193                 }
00194             }
00195
00196             // running error code
00197             RC ret = SUCCESS;
00198
00199             // make initial guess if the user did not make one
00200             if( nnz( pr ) != n ) {
00201                 ret = set( pr, static_cast< IOType >( 1 ) / static_cast< IOType >( n ) );
00202                 assert( ret == SUCCESS );
00203             }
00204
00205             // initialise all temporary vectors to default dense values
00206             ret = ret ? ret : set( pr_nextnext, zero );
00207             assert( ret == SUCCESS );
00208
00209             // calculate row sums
00210             Semiring<
00211                 operators::add< IOType >,
00212                 operators::left_assign_if< IOType, bool, IOType >,
00213                 identities::zero,
00214                 identities::logical_true
00215             > pattern_ring;
00216
00217             ret = ret ? ret : set( pr_next, 1 ); // abuses pr_next as temporary vector
00218             ret = ret ? ret : set( row_sum, 0 );
00219             ret = ret ? ret :
00220                 vxm< descr | descriptors::dense | descriptors::transpose_matrix >(
00221                     row_sum, pr_next, L, pattern_ring
00222                 );
00223             // pr_next is now free for further use
00224             assert( ret == SUCCESS );
00225
00226 #ifdef _DEBUG
00227             std::cout « "Prelude to iteration 0:\n";
00228             (void) eWiseLambda(
00229                 [ &row_sum, &pr_next, &pr ]( const size_t i ) {
00230                     #pragma omp critical
```

```
00231                     {
00232                         std::cout « i « ": " « row_sum[ i ] « "\t" « pr[ i ] « "\t"
00233                             « pr_next[ i ] « "\n";
00234                     }
00235                 },
00236                 pr, pr_next, row_sum );
00237 #endif
00238
00239             // calculate in-place the inverse of row sums, and keep zero if dangling
00240             // this eWiseLambda is always supported since alpha is read-only
00241             ret = ret ? ret : eWiseLambda(
00242                     [ &row_sum, &alpha, &zero ]( const size_t i ) {
00243                         assert( row_sum[ i ] >= zero );
00244                         if( row_sum[ i ] > zero ) {
00245                             row_sum[ i ] = alpha / row_sum[ i ];
00246                         }
00247                     },
00248                     row_sum
00249                 );
00250                 assert( ret == SUCCESS );
00251
00252 #ifdef _DEBUG
00253             std::cout « "Row sum array:\n";
00254             (void) eWiseLambda(
00255                 [ &row_sum ]( const size_t i ) {
00256                     #pragma omp critical
00257                     std::cout « i « ": " « row_sum[ i ] « "\n";
00258                 }, row_sum
00259             );
00260 #endif
00261
00262             // some control variables
00263             size_t iter = 0; //#iterations, initalise to zero
00264             IOType dangling = zero; // used for caching the contribution of random jumps
00265                                     // from dangling nodes
00266             IOType residual = zero; // declare residual (computed inside the do-while
00267                                     // loop)
00268
00269             // main loop
00270             do {
00271                 // reset iteration-local values
00272                 residual = dangling = 0;
00273
00274 #ifdef _DEBUG
00275                 std::cout « "Current PR array:\n";
00276                 (void) eWiseLambda(
00277                         [ &pr ]( const size_t i ) {
00278                             #pragma omp critical
00279                             if( i < 8 ) {
00280                                 std::cout « i « ": " « pr[ i ] « "\n";
00281                             }
00282                         }, pr
00283                     );
00284 #endif
00285
00286                 // calculate dangling factor and do scaling
00287                 if( ret == SUCCESS ) {
00288                     // can we reduce via lambdas?
00289                     if( Properties<>::writableCaptured ) {
00290                         // yes we can, so save one unnecessary stream on pr
00291                         ret = eWiseLambda(
00292                                 [ &pr_next, &row_sum, &dangling, &pr ]( const size_t i ) {
00293                                     // calculate dangling contribution
00294                                     if( row_sum[ i ] == 0 ) {
00295                                         dangling += pr[ i ];
00296                                         pr_next[ i ] = 0;
00297                                     } else {
00298                                         // pre-scale input
00299                                         pr_next[ i ] = pr[ i ] * row_sum[ i ];
00300                                     }
00301                                 }, row_sum, pr, pr_next
00302                             );
00303                         // allreduce dangling factor
00304                         assert( ret == SUCCESS );
00305                         ret = ret ? ret : grb::collectives<>::allreduce(
00306                             dangling,
00307                             grb::operators::add< double >()
00308                         );
00309                         assert( ret == SUCCESS );
00310                     } else {
00311                         // otherwise we have to handle the reduction separately
00312                         ret = foldl< grb::descriptors::invert_mask >(
00313                             dangling, pr, row_sum, addM
00314                         );
00315                         assert( ret == SUCCESS );
00316
00317                         // separately from the element-wise multiplication here
```

```
00318                         ret = ret ? ret : set( pr_next, 0 );
00319                         ret = ret ? ret : eWiseApply(
00320                                 pr_next, pr, row_sum,
00321                                 grb::operators::mul< double >()
00322                             );
00323                         assert( ret == SUCCESS );
00324                     }
00325                 }
00326
00327 #if defined _DEBUG && defined _GRB_WITH_LPF
00328                 for( size_t dbg = 0; dbg < spmd<>::nprocs(); ++dbg ) {
00329                     const auto s = spmd<>::pid();
00330                     if( dbg == s ) {
00331                         std::cout « "Next PR array (under construction):\n";
00332                         eWiseLambda(
00333                             [ &pr_next, s ]( const size_t i ) {
00334                                 #pragma omp critical
00335                                 if( i < 10 ) {
00336                                     std::cout « i « ", " « s « ": " « pr_next[ i ] « "\n";
00337                                 }
00338                             },
00339                             pr_next );
00340                     }
00341                     spmd<>::sync();
00342                 }
00343 #endif
00344
00345                 if( ret == SUCCESS ) {
00346 #ifdef _DEBUG
00347                     std::cout « s « ": dangling (1) = " « dangling « "\n";
00348 #endif
00349
00350                     // complete dangling factor
00351                     dangling = ( alpha * dangling + 1 - alpha ) / static_cast< IOType >( n );
00352
00353 #ifdef _DEBUG
00354                     std::cout « s « ": dangling (2) = " « dangling « "\n";
00355 #endif
00356                 }
00357
00358                 // multiply with row-normalised link matrix (no change to dangling rows)
00359                 // note that the later eWiseLambda requires the output be dense
00360                 ret = ret ? ret : set( pr_nextnext, 0 ); assert( ret == SUCCESS );
00361                 ret = ret ? ret : vxm< descr >( pr_nextnext, pr_next, L, realRing );
00362                 assert( ret == SUCCESS );
00363                 assert( n == grb::nnz( pr_nextnext ) );
00364
00365 #if defined _DEBUG && defined _GRB_WITH_LPF
00366                 for( size_t dbg = 0; dbg < spmd<>::nprocs(); ++dbg ) {
00367                     if( dbg == s ) {
00368                         std::cout « s « ": nextnext PR array (after vxm):\n";
00369                         (void) eWiseLambda(
00370                             [ &pr_nextnext, s ]( const size_t i ) {
00371                                 #pragma omp critical
00372                                 if( i < 10 )
00373                                     std::cout « i « ", " « s « ": " « pr_nextnext[ i ] « "\n";
00374                             }, pr_nextnext
00375                         );
00376                     }
00377                     (void) spmd<>::sync();
00378                 }
00379                 for( size_t k = 0; k < spmd<>::nprocs(); ++k ) {
00380                     if( spmd<>::pid() == k ) {
00381                         std::cout « "old pr \t scaled input \t alpha * pr * H at PID "
00382                             « k « "\n";
00383                         (void) eWiseLambda(
00384                             [ &pr, &pr_next, &pr_nextnext ]( const size_t i ) {
00385                                 #pragma omp critical
00386                                 {
00387                                     std::cout « pr[ i ] « "\t" « pr_next[ i ] « "\t"
00388                                         « pr_nextnext[ i ] « "\n";
00389                                 }
00390                             }, pr, pr_next, pr_nextnext
00391                         );
00392                     }
00393                     (void) spmd<>::sync();
00394                 }
00395 #endif
00396
00397                 // calculate & normalise new pr and calculate residual
00398                 if( ret == SUCCESS ) {
00399                     // can we reduce via lambdas?
00400                     if( grb::Properties<>::writableCaptured ) {
00401                         // yes, we can. So update pr[ i ] and calculate residual simultaneously
00402                         ret = eWiseLambda(
00403                             [ &pr, &pr_nextnext, &dangling, &residual, &zero ]( const size_t i ) {
00404                                 // cache old pagerank vector
```

```
00405                                 const IOType oldval = pr[ i ];
00406                                 // set new pagerank vector
00407                                 pr[ i ] = pr_nextnext[ i ] + dangling;
00408                                 // update residual
00409                                 if( oldval > pr[ i ] ) {
00410                                     residual += oldval - pr[ i ];
00411                                 } else {
00412                                     residual += pr[ i ] - oldval;
00413                                 }
00414                                 pr_nextnext[ i ] = zero;
00415                             }, pr, pr_nextnext
00416                         );
00417                         // reduce process-local residual
00418                         assert( ret == SUCCESS );
00419                         if( ret == SUCCESS ) {
00420                             ret = grb::collectives<>::allreduce(
00421                                 residual,
00422                                 grb::operators::add< double >()
00423                             );
00424                         }
00425                         assert( ret == SUCCESS );
00426                     } else {
00427                         // we cannot reduce via lambdas, so calculate new pr vector
00428                         ret = foldl< descriptors::dense >( pr_nextnext, dangling, addM );
00429                         assert( ret == SUCCESS );
00430                         // do a dot product under the one-norm "ring"
00431                         if( ret == SUCCESS ) {
00432                             residual = zero;
00433                             ret = dot< descriptors::dense >(
00434                                 residual,
00435                                 pr, pr_nextnext,
00436                                 addM, grb::operators::abs_diff< IOType >()
00437                             );
00438                             assert( ret == SUCCESS );
00439                         }
00440                         if( ret == SUCCESS ) {
00441                             // next pr vector becomes current pr vector
00442                             std::swap( pr, pr_nextnext );
00443                         }
00444                     }
00445                 }
00446
00447                 // update iteration count
00448                 ++iter;
00449
00450                 // check convergence
00451                 if( conv != zero && residual <= conv ) { break; }
00452
00453 #ifdef _DEBUG
00454                 if( grb::spmd<>::pid() == 0 ) {
00455                     std::cout << "Iteration " << iter << ", "
00456                         << "residual = " << residual << std::endl;
00457                 }
00458 #endif
00459
00460             } while( ret == SUCCESS && iter < max );
00461
00462             // check if the user requested any stats, and output if yes
00463             if( iterations != nullptr ) {
00464                 *iterations = iter;
00465             }
00466             if( quality != nullptr ) {
00467                 *quality = residual;
00468             }
00469
00470             // return the appropriate exit code
00471             if( ret != SUCCESS ) {
00472                 if( spmd<>::pid() == 0 ) {
00473                     std::cerr << "Error while running simple pagerank algorithm: "
00474                         << toString( ret ) << "\n";
00475                 }
00476                 return ret;
00477             } else if( residual <= conv ) {
00478 #ifdef _DEBUG
00479                 if( spmd<>::pid() == 0 ) {
00480                     std::cerr << "Info: simple pagerank converged after " << iter
00481                         << " iterations.\n";
00482                 }
00483 #endif
00484                 return SUCCESS; // converged!
00485             } else {
00486 #ifdef _DEBUG
00487                 if( spmd<>::pid() == 0 ) {
00488                     std::cout << "Info: simple pagerank did not converge after "
00489                         << iter << " iterations.\n";
00490                 }
00491 #endif
```

```
00492                 return FAILED; // not converged
00493             }
00494         }
00495
00496     } // namespace algorithms
00497
00498 } // namespace grb
00499
00500 #endif // end _H_GRB_PAGERANK
00501
```

10.25 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.

10.25.1 Detailed Description

Implements (non-batched) sparse neural network inference.

Author

Aristeidis Mastoras

10.26 sparse_nn_single_inference.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00026 #ifndef _H_GRB_ALGORITHMS_SPARSE_NN_SINGLE_INFERENCE
00027 #define _H_GRB_ALGORITHMS_SPARSE_NN_SINGLE_INFERENCE
00028
00029 #include <limits>
00030 #include <graphblas.hpp>
00031
00032
00033 namespace grb {
00034
00035     namespace algorithms {
00036
00037         namespace internal {
00038
00047             template<
00048                 Descriptor descr,
00049                 bool thresholded, typename ThresholdType,
00050                 typename IOType, typename WeightType, typename BiasType,
00051                 class ReluMonoid, class Ring, class MinMonoid
00052             >
00053             grb::RC sparse_nn_single_inference(
00054                 grb::Vector< IOType > &out,
00055                 const grb::Vector< IOType > &in,
00056                 const std::vector< grb::Matrix< WeightType > > &layers,
00057                 const std::vector< BiasType > &biases,
00058                 const ThresholdType threshold,
00059                 grb::Vector< IOType > &temp,
00060                 const ReluMonoid &relu,
00061                 const MinMonoid &min,
00062                 const Ring &ring
00063             ) {
00064                 static_assert( !(descr & descriptors::no_casting) ||
00065                     (
00066                         std::is_same< IOType, WeightType >::value &&
00067                         std::is_same< IOType, BiasType >::value
00068                     ), "Input containers have different domains even though the no_casting"
00069                     "descriptor was given"
00070                 );
00071
00072                 const size_t num_layers = layers.size();
00073
00074                 // run-time checks
00075                 {
00076                     const size_t n = grb::size( out );
00077                     if( num_layers == 0 ) {
00078                         return ILLEGAL;
00079                     }
00080                     if( biases.size() != num_layers ) {
00081                         return ILLEGAL;
00082                     }
00083                     if( grb::size( in ) != grb::nrows( ( layers[ 0 ] ) ) ||
00084                         grb::size( out ) != grb::ncols( ( layers[ num_layers - 1 ] ) ) ||
00085                         grb::size( out ) != grb::size( temp )
00086                     ) {
00087                         return MISMATCH;
00088                     }
00089                     for( size_t i = 1; i < num_layers; ++i ) {
00090                         if( grb::ncols( ( layers[ i - 1 ] ) ) != grb::nrows( ( layers[ i ] ) ) ) {
00091                             return MISMATCH;
00092                         }
00093                     }
00094                     for( size_t i = 0; i < num_layers; ++i ) {
00095                         if( grb::ncols( ( layers[ i ] ) ) != grb::nrows( ( layers[ i ] ) ) ) {
00096                             return ILLEGAL;
00097                         }
00098                     }
```

```
00099                     assert( n == grb::size( in ) );
00100                     assert( n == grb::size( temp ) );
00101                     if( grb::capacity( out ) != n ) {
00102                         return ILLEGAL;
00103                     }
00104                     if( grb::capacity( temp ) != n ) {
00105                         return ILLEGAL;
00106                     }
00107                 }
00108 
00109                 grb::RC ret = SUCCESS;
00110 
00111     /*
00112                 // this is a correct implementation that does not unroll the first and the last
      iterations
00113                 // we do not use it because it requires setting the input vector to the output vector
00114                 // which results in copying data for 2*n elements
00115 
00116                 ret = grb::set( out, in );
00117 
00118                 for( size_t i = 1; ret == SUCCESS && i < num_layers ; ++i ) {
00119 
00120                     std::swap( out, temp );
00121                     ret = ret ? ret : grb::set( out, 0 );
00122                     ret = ret ? ret : grb::vxm( out, temp, *(layers[ i - 1 ]), ring );
00123                     ret = ret ? ret : grb::foldl< descriptors::dense >( out, biases[ i ],
      ring.getAdditiveMonoid() );
00124                     ret = ret ? ret : grb::foldl< descriptors::dense >( out, 0, relu );
00125                     if( thresholded ) {
00126                         ret = ret ? ret : grb::foldl< descriptors::dense >( out, threshold, min );
00127                     }
00128                 }
00129     */
00130 
00131                 ret = grb::set( out, 0 ); assert( ret == SUCCESS );
00132 
00133                 ret = ret ? ret : grb::vxm( out, in, ( layers[ 0 ] ), ring );
00134                 assert( ret == SUCCESS );
00135 
00136                 ret = ret ? ret : grb::foldl< descriptors::dense >(
00137                     out, biases[ 1 ], ring.getAdditiveMonoid()
00138                 ); assert( ret == SUCCESS );
00139 
00140                 for( size_t i = 1; ret == SUCCESS && i < num_layers - 1; ++i ) {
00141 
00142                     ret = ret ? ret : grb::foldl< descriptors::dense >( out, 0, relu );
00143                     assert( ret == SUCCESS );
00144 
00145                     if( thresholded ) {
00146                         ret = ret ? ret : grb::foldl< descriptors::dense >( out, threshold, min );
00147                         assert( ret == SUCCESS );
00148                     }
00149 
00150                     if( ret == SUCCESS ) {
00151                         std::swap( out, temp );
00152                     }
00153 
00154                     ret = grb::set( out, 0 );
00155                     assert( ret == SUCCESS );
00156 
00157                     ret = ret ? ret : grb::vxm< descriptors::dense >(
00158                         out, temp, ( layers[ i ] ), ring
00159                     ); assert( ret == SUCCESS );
00160 
00161                     ret = ret ? ret : grb::foldl< descriptors::dense >(
00162                         out, biases[ i + 1 ], ring.getAdditiveMonoid()
00163                     ); assert( ret == SUCCESS );
00164                 }
00165 
00166                 ret = ret ? ret : grb::foldl< descriptors::dense >( out, 0, relu );
00167                 assert( ret == SUCCESS );
00168 
00169                 if( thresholded ) {
00170                     ret = ret ? ret : grb::foldl< descriptors::dense >( out, threshold, min );
00171                     assert( ret == SUCCESS );
00172                 }
00173 
00174                 return ret;
00175             }
00176 
00177         } // end namespace "grb::internal"
00178 
00257         template< Descriptor descr = descriptors::no_operation,
00258             typename IOType,
00259             typename WeightType,
00260             typename BiasType,
00261             class ReluMonoid = Monoid<
```

```
00262                 grb::operators::relu< IOType >,
00263                 grb::identities::negative_infinity
00264             >,
00265             class Ring = Semiring<
00266                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
00267                 grb::identities::zero, grb::identities::one
00268             >
00269         >
00270         grb::RC sparse_nn_single_inference(
00271             grb::Vector< IOType > &out,
00272             const grb::Vector< IOType > &in,
00273             const std::vector< grb::Matrix< WeightType > > &layers,
00274             const std::vector< BiasType > &biases,
00275             grb::Vector< IOType > &temp,
00276             const ReluMonoid &relu = ReluMonoid(),
00277             const Ring &ring = Ring()
00278         ) {
00279             static_assert( !(descr & descriptors::no_casting) ||
00280                 (
00281                     std::is_same< IOType, WeightType >::value &&
00282                     std::is_same< IOType, BiasType >::value
00283                 ), "Input containers have different domains even though the no_casting "
00284                 "descriptor was given" );
00285             Monoid<
00286                 grb::operators::min< IOType >, grb::identities::infinity
00287             > dummyTresholdMonoid;
00288             return internal::sparse_nn_single_inference<
00289                 descr, false, double
00290             > (
00291                 out, in, layers,
00292                 biases, 0.0,
00293                 temp,
00294                 relu, dummyTresholdMonoid, ring
00295             );
00296         }
00297
00383         template< Descriptor descr = descriptors::no_operation,
00384             typename IOType,
00385             typename WeightType,
00386             typename BiasType,
00387             typename ThresholdType = IOType,
00388             class MinMonoid = Monoid<
00389                 grb::operators::min< IOType >, grb::identities::infinity
00390             >,
00391             class ReluMonoid = Monoid<
00392                 grb::operators::relu< IOType >,
00393                 grb::identities::negative_infinity
00394             >,
00395             class Ring = Semiring<
00396                 grb::operators::add< IOType >, grb::operators::mul< IOType >,
00397                 grb::identities::zero, grb::identities::one
00398             >
00399         >
00400         grb::RC sparse_nn_single_inference(
00401             grb::Vector< IOType > &out,
00402             const grb::Vector< IOType > &in,
00403             const std::vector< grb::Matrix< WeightType > > &layers,
00404             const std::vector< BiasType > &biases,
00405             const ThresholdType threshold,
00406             grb::Vector< IOType > &temp,
00407             const ReluMonoid &relu = ReluMonoid(),
00408             const MinMonoid &min = MinMonoid(),
00409             const Ring &ring = Ring()
00410         ) {
00411             static_assert( !(descr & descriptors::no_casting) ||
00412                 (
00413                     std::is_same< IOType, WeightType >::value &&
00414                     std::is_same< IOType, BiasType >::value
00415                 ), "Input containers have different domains even though the no_casting "
00416                 "descriptor was given" );
00417             return internal::sparse_nn_single_inference<
00418                 descr, true
00419             > (
00420                 out, in, layers,
00421                 biases, threshold,
00422                 temp,
00423                 relu, min, ring
00424             );
00425         }
00426
00427     } // namespace algorithms
00428
00429 } // end namespace grb
00430
00431 #endif // end _H_GRB_ALGORITHMS_SPARSE_NN_SINGLE_INFERENCE
00432
```

10.27 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.

10.27.1 Detailed Description

Implements a simple matrix spy algorithm.

Author

A. N. Yzelman

10.28 spy.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00026 #ifndef _H_GRB_ALGORITHMS_SPY
00027 #define _H_GRB_ALGORITHMS_SPY
00028
00029 #include <type_traits>
```

```
00030 #include <vector>
00031
00032 #include <graphblas.hpp>
00033
00034
00035 namespace grb {
00036
00037     namespace algorithms {
00038
00039         namespace internal {
00040
00047             template< bool normalize, typename IOType, typename InputType >
00048             RC spy_from_bool_or_void_input(
00049                 grb::Matrix< IOType > &out, const grb::Matrix< InputType > &in,
00050                 const size_t m, const size_t n,
00051                 const size_t small_m, const size_t small_n
00052             ) {
00053                 static_assert( std::is_same< InputType, bool >::value ||
00054                     std::is_same< InputType, void >::value,
00055                     "Error in call to internal::spy_from_bool_or_void_input"
00056                 );
00057
00058                 // Q must be n by small_n
00059                 grb::Matrix< unsigned char > Q( n, small_n );
00060                 grb::RC ret = grb::resize( Q, n );
00061                 // TODO FIXME use repeating + auto-incrementing iterators
00062                 std::vector< size_t > I, J;
00063                 std::vector< unsigned char > V;
00064                 const double n_sample = static_cast< double >(n) /
00065                     static_cast< double >(small_n);
00066                 for( size_t i = 0; i < n; ++i ) {
00067                     I.push_back( i );
00068                     J.push_back( static_cast< double >(i) / n_sample );
00069                     V.push_back( 1 );
00070                 }
00071                 ret = grb::buildMatrixUnique(
00072                     Q, &(I[0]), &(J[0]), &(V[0]), n,
00073                     grb::SEQUENTIAL
00074                 );
00075
00076                 // P must be small_m by m
00077                 grb::Matrix< unsigned char > P( small_m, m );
00078                 if( ret == SUCCESS ) {
00079                     ret = grb::resize( P, m );
00080                     // TODO FIXME use repeating + auto-incrementing iterators
00081                     std::vector< size_t > I, J;
00082                     std::vector< unsigned char > V;
00083                     const double m_sample = static_cast< double >(m) /
00084                         static_cast< double >(small_m);
00085                     for( size_t i = 0; i < m; ++i ) {
00086                         I.push_back( static_cast< double >(i) / m_sample );
00087                         J.push_back( i );
00088                         V.push_back( 1 );
00089                     }
00090                     ret = grb::buildMatrixUnique(
00091                         P, &(I[0]), &(J[0]), &(V[0]), m,
00092                         grb::SEQUENTIAL
00093                     );
00094                 }
00095
00096                 // tmp must be m by small_n OR small_m by n
00097                 if( ret == SUCCESS && m - small_m > n - small_n ) {
00098                     grb::Semiring<
00099                         grb::operators::add< size_t >,
00100                         grb::operators::left_assign_if< size_t, bool, size_t >,
00101                         grb::identities::zero,
00102                         grb::identities::logical_true
00103                     > leftAssignAndAdd;
00104                     grb::Matrix< size_t > tmp( small_m, n );
00105                     ret = ret ? ret : grb::mxm( tmp, P, in, leftAssignAndAdd, RESIZE );
00106                     ret = ret ? ret : grb::mxm( tmp, P, in, leftAssignAndAdd, EXECUTE );
00107                     ret = ret ? ret : grb::mxm( out, tmp, Q, leftAssignAndAdd, RESIZE );
00108                     ret = ret ? ret : grb::mxm( out, tmp, Q, leftAssignAndAdd, EXECUTE );
00109                 } else {
00110                     grb::Semiring<
00111                         grb::operators::add< size_t >,
00112                         grb::operators::right_assign_if< bool, size_t, size_t >,
00113                         grb::identities::zero,
00114                         grb::identities::logical_true
00115                     > rightAssignAndAdd;
00116                     grb::Matrix< size_t > tmp( m, small_n );
00117                     ret = ret ? ret : grb::mxm( tmp, in, Q, rightAssignAndAdd, RESIZE );
00118                     ret = ret ? ret : grb::mxm( tmp, in, Q, rightAssignAndAdd, EXECUTE );
00119                     ret = ret ? ret : grb::mxm( out, P, tmp, rightAssignAndAdd, RESIZE );
00120                     ret = ret ? ret : grb::mxm( out, P, tmp, rightAssignAndAdd, EXECUTE );
00121                 }
00122
```

```
00123                 if( ret == SUCCESS && normalize ) {
00124                     ret = grb::eWiseLambda( [] (const size_t, const size_t, IOType &v ) {
00125                         assert( v > 0 );
00126                         v = static_cast< IOType >( 1 ) / v;
00127                     }, out );
00128                 }
00129 
00130                 return ret;
00131             }
00132 
00133         }
00134 
00194         template<
00195             bool normalize = false,
00196             typename IOType, typename InputType
00197         >
00198         RC spy( grb::Matrix< IOType > &out, const grb::Matrix< InputType > &in ) {
00199             static_assert( !normalize || std::is_floating_point< IOType >::value,
00200                 "When requesting a normalised spy plot, the data type must be "
00201                 "floating-point"
00202             );
00203 
00204             const size_t m = grb::nrows( in );
00205             const size_t n = grb::ncols( in );
00206             const size_t small_m = grb::nrows( out );
00207             const size_t small_n = grb::ncols( out );
00208 
00209             // runtime checks and shortcuts
00210             if( small_m > m ) { return ILLEGAL; }
00211             if( small_n > n ) { return ILLEGAL; }
00212             if( small_m == m && small_n == n ) {
00213                 return grb::set< grb::descriptors::structural >( out, in, 1 );
00214             }
00215 
00216             grb::RC ret = grb::clear( out );
00217 
00218             grb::Matrix< bool > tmp( m, n );
00219             ret = ret ? ret : grb::resize( tmp, grb::nnz( in ) );
00220             ret = ret ? ret : grb::set< grb::descriptors::structural >( tmp, in, true );
00221             ret = ret ? ret : grb::algorithms::internal::template
00222                 spy_from_bool_or_void_input< normalize >(
00223                     out, tmp, m, n, small_m, small_n
00224                 );
00225 
00226             return ret;
00227         }
00228 
00230         template< bool normalize = false, typename IOType >
00231         RC spy( grb::Matrix< IOType > &out, const grb::Matrix< bool > &in ) {
00232             static_assert( !normalize || std::is_floating_point< IOType >::value,
00233                 "When requesting a normalised spy plot, the data type must be "
00234                 "floating-point" );
00235 
00236             const size_t m = grb::nrows( in );
00237             const size_t n = grb::ncols( in );
00238             const size_t small_m = grb::nrows( out );
00239             const size_t small_n = grb::ncols( out );
00240 
00241             // runtime checks and shortcuts
00242             if( small_m > m ) { return ILLEGAL; }
00243             if( small_n > n ) { return ILLEGAL; }
00244             if( small_m == m && small_n == n ) {
00245                 return grb::set< grb::descriptors::structural >( out, in, 1 );
00246             }
00247 
00248             grb::RC ret = grb::clear( out );
00249 
00250             ret = ret ? ret : grb::algorithms::internal::template
00251                 spy_from_bool_or_void_input< normalize >(
00252                     out, in, m, n, small_m, small_n
00253                 );
00254 
00255             return ret;
00256         }
00257 
00259         template< bool normalize = false, typename IOType >
00260         RC spy( grb::Matrix< IOType > &out, const grb::Matrix< void > &in ) {
00261             static_assert( !normalize || std::is_floating_point< IOType >::value,
00262                 "When requesting a normalised spy plot, the data type must be "
00263                 "floating-point"
00264             );
00265 
00266             const size_t m = grb::nrows( in );
00267             const size_t n = grb::ncols( in );
00268             const size_t small_m = grb::nrows( out );
00269             const size_t small_n = grb::ncols( out );
00270 
```

```
00271             // runtime checks and shortcuts
00272             if( small_m > m ) { return ILLEGAL; }
00273             if( small_n > n ) { return ILLEGAL; }
00274             if( small_m == m && small_n == n ) {
00275                 return grb::set< grb::descriptors::structural >( out, in, 1 );
00276             }
00277
00278             grb::RC ret = grb::clear( out );
00279
00280             ret = ret ? ret : grb::algorithms::internal::template
00281                 spy_from_bool_or_void_input< normalize >(
00282                     out, in, m, n, small_m, small_n
00283                 );
00284
00285             return ret;
00286         }
00287
00288     } // end namespace "grb::algorithms"
00289
00290 } // end namespace "grb"
00291
00292 #endif // _H_GRB_ALGORITHMS_SPY
00293
```

10.29 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 , **shmem1D** ,
  NUMA1D , **GENERIC_BSP** , BSP1D , **doublyBSP1D** ,
  BSP2D , **autoBSP** , **optBSP** , hybrid ,
  hybridSmall , **hybridMid** , **hybridLarge** , **minFootprint** ,
  banshee , **banshee_ssr** }

  A collection of all backends.

10.29.1 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

10.30 backends.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00029 #ifndef _H_GRB_BACKENDS
00030 #define _H_GRB_BACKENDS
00031
00032
00033 namespace grb {
00034
00046     enum Backend {
00047
00052         reference,
00053
00059         reference_omp,
00060
00067         hyperdags,
00068
00075         shmem1D,
00076
00084         NUMA1D,
00085
00092         GENERIC_BSP,
00093
00104         BSP1D,
00105
00114         doublyBSP1D,
00115
00124         BSP2D,
00125
00136         autoBSP,
00137
00145         optBSP,
00146
00154         hybrid,
00155
00163         hybridSmall,
00164
00174         hybridMid,
00175
00188         hybridLarge,
00189
00195         minFootprint,
00196
00201         banshee,
00202
00211         banshee_ssr
00212
00213     };
00214
00215 } // namespace grb
00216
00217 #endif
00218
```

10.31 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.

10.31.1 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

10.32 benchmark.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00028 #ifndef _H_GRB_BENCH_BASE
00029 #define _H_GRB_BENCH_BASE
00030
00031 #include <chrono>
00032 #include <ios>
00033 #include <limits>
00034 #include <string>
00035
00036 #include <graphblas/backends.hpp>
00037 #include <graphblas/ops.hpp>
00038 #include <graphblas/rc.hpp>
00039 #include <graphblas/utils.hpp>
00040 #include <graphblas/utils/TimerResults.hpp>
00041
00042 #include "collectives.hpp"
00043 #include "config.hpp"
00044 #include "exec.hpp"
00045
00046 #ifndef _GRB_NO_STDIO
00047  #include <iostream>
00048 #endif
```

```
00049
00050 #ifndef _GRB_NO_EXCEPTIONS
00051  #include <stdexcept>
00052 #endif
00053
00054 #include <math.h>
00055
00056
00082 namespace grb {
00083
00084     namespace internal {
00085
00091         class BenchmarkerBase {
00092
00093             protected:
00094
00095 #ifndef _GRB_NO_STDIO
00104                 static void printTimeSinceEpoch( const bool printHeader = true ) {
00105                     const auto now = std::chrono::system_clock::now();
00106                     const auto since = now.time_since_epoch();
00107                     if( printHeader ) {
00108                         std::cout « "Time since epoch (in ms.): ";
00109                     }
00110                     std::cout « std::chrono::duration_cast<
00111                             std::chrono::milliseconds
00112                         >( since ).count() « "\n";
00113                 }
00114 #endif
00115
00119                 static void benchmark_calc_inner(
00120                     const size_t loop,
00121                     const size_t total,
00122                     grb::utils::TimerResults &inner_times,
00123                     grb::utils::TimerResults &total_times,
00124                     grb::utils::TimerResults &min_times,
00125                     grb::utils::TimerResults &max_times,
00126                     grb::utils::TimerResults * sdev_times
00127                 ) {
00128                     inner_times.normalize( total );
00129                     total_times.accum( inner_times );
00130                     min_times.min( inner_times );
00131                     max_times.max( inner_times );
00132                     sdev_times[ loop ] = inner_times;
00133                 }
00134
00138                 static void benchmark_calc_outer(
00139                     const size_t total,
00140                     grb::utils::TimerResults &total_times,
00141                     grb::utils::TimerResults &min_times,
00142                     grb::utils::TimerResults &max_times,
00143                     grb::utils::TimerResults * sdev_times,
00144                     const size_t pid
00145                 ) {
00146                     total_times.normalize( total );
00147                     grb::utils::TimerResults sdev;
00148                     // compute standard dev of average times, leaving sqrt calculation until
00149                     // the output of the values
00150                     sdev.set( 0 );
00151                     for( size_t i = 0; i < total; i++ ) {
00152                         double diff = sdev_times[ i ].io - total_times.io;
00153                         sdev.io += diff * diff;
00154                         diff = sdev_times[ i ].preamble - total_times.preamble;
00155                         sdev.preamble += diff * diff;
00156                         diff = sdev_times[ i ].useful - total_times.useful;
00157                         sdev.useful += diff * diff;
00158                         diff = sdev_times[ i ].postamble - total_times.postamble;
00159                         sdev.postamble += diff * diff;
00160                     }
00161                     // unbiased normalisation of the standard deviation
00162                     sdev.normalize( total - 1 );
00163
00164 #ifndef _GRB_NO_STDIO
00165                     // output results
00166                     if( pid == 0 ) {
00167                         std::cout « "Overall timings (io, preamble, useful, postamble):\n"
00168                             « std::scientific;
00169                         std::cout « "Avg: " « total_times.io « ", " « total_times.preamble
00170                             « ", " « total_times.useful « ", " « total_times.postamble « "\n";
00171                         std::cout « "Min: " « min_times.io « ", " « min_times.preamble « ", "
00172                             « min_times.useful « ", " « min_times.postamble « "\n";
00173                         std::cout « "Max: " « max_times.io « ", " « max_times.preamble « ", "
00174                             « max_times.useful « ", " « max_times.postamble « "\n";
00175                         std::cout « "Std: " « sqrt( sdev.io ) « ", " « sqrt( sdev.preamble )
00176                             « ", " « sqrt( sdev.useful ) « ", " « sqrt( sdev.postamble ) « "\n";
00177  #if __GNUC__ > 4
00178                         std::cout « std::defaultfloat;
00179  #endif
```

```
00180                         printTimeSinceEpoch();
00181                     }
00182 #else
00183                     // we ran the benchmark, but may not have a way to output it in this case
00184                     // this currently only is touched by the #grb::banshee backend, which
00185                     // provides other timing mechanisms.
00186                     (void) min_times;
00187                     (void) max_times;
00188                     (void) pid;
00189 #endif
00190                 }
00191
00213                 template<
00214                     typename U,
00215                     enum Backend implementation = config::default_backend
00216                 >
00217                 static RC benchmark(
00218                     void ( *alp_program )( const void *, const size_t, U & ),
00219                     const void * data_in,
00220                     const size_t in_size,
00221                     U &data_out,
00222                     const size_t inner,
00223                     const size_t outer,
00224                     const size_t pid
00225                 ) {
00226                     const double inf = std::numeric_limits< double >::infinity();
00227                     grb::utils::TimerResults total_times, min_times, max_times;
00228                     grb::utils::TimerResults * sdev_times =
00229                         new grb::utils::TimerResults[ outer ];
00230                     total_times.set( 0 );
00231                     min_times.set( inf );
00232                     max_times.set( 0 );
00233
00234                     // outer loop
00235                     for( size_t out = 0; out < outer; ++out ) {
00236                         grb::utils::TimerResults inner_times;
00237                         inner_times.set( 0 );
00238
00239                         // inner loop
00240                         for( size_t in = 0; in < inner; in++ ) {
00241                             data_out.times.set( 0 );
00242                             ( *alp_program )( data_in, in_size, data_out );
00243                             grb::collectives< implementation >::reduce(
00244                                 data_out.times.io, 0, grb::operators::max< double >() );
00245                             grb::collectives< implementation >::reduce(
00246                                 data_out.times.preamble, 0, grb::operators::max< double >() );
00247                             grb::collectives< implementation >::reduce(
00248                                 data_out.times.useful, 0, grb::operators::max< double >() );
00249                             grb::collectives< implementation >::reduce(
00250                                 data_out.times.postamble, 0, grb::operators::max< double >() );
00251                             inner_times.accum( data_out.times );
00252                         }
00253
00254                         // calculate performance stats
00255                         benchmark_calc_inner( out, inner, inner_times, total_times, min_times,
00256                             max_times, sdev_times );
00257
00258 #ifndef _GRB_NO_STDIO
00259                         // give experiment output line
00260                         if( pid == 0 ) {
00261                             std::cout « "Outer iteration #" « out « " timings (io, preamble, "
00262                                 « "useful, postamble, time since epoch): ";
00263                             std::cout « inner_times.io « ", " « inner_times.preamble « ", "
00264                                 « inner_times.useful « ", " « inner_times.postamble « ", ";
00265                             printTimeSinceEpoch( false );
00266                         }
00267 #endif
00268
00269                         // pause for next outer loop
00270                         if( sleep( 1 ) != 0 ) {
00271 #ifndef _GRB_NO_STDIO
00272                             std::cerr « "Sleep interrupted, assume benchmark is unreliable; "
00273                                 « "exiting.\n";
00274 #endif
00275                             abort();
00276                         }
00277                     }
00278
00279                     // calculate performance stats
00280                     benchmark_calc_outer( outer, total_times, min_times, max_times, sdev_times,
00281                         pid );
00282                     delete [] sdev_times;
00283
00284                     return SUCCESS;
00285                 }
00286
00308                 template<
```

```
00309                     typename T, typename U,
00310                     enum Backend implementation = config::default_backend
00311                 >
00312                 static RC benchmark(
00313                     void ( *alp_program )( const T &, U & ),
00314                     const T &data_in,
00315                     U &data_out,
00316                     const size_t inner,
00317                     const size_t outer,
00318                     const size_t pid
00319                 ) {
00320                     const double inf = std::numeric_limits< double >::infinity();
00321                     grb::utils::TimerResults total_times, min_times, max_times;
00322                     grb::utils::TimerResults * sdev_times =
00323                         new grb::utils::TimerResults[ outer ];
00324                     total_times.set( 0 );
00325                     min_times.set( inf );
00326                     max_times.set( 0 );
00327
00328                     // outer loop
00329                     for( size_t out = 0; out < outer; ++out ) {
00330                         grb::utils::TimerResults inner_times;
00331                         inner_times.set( 0 );
00332
00333                         // inner loop
00334                         for( size_t in = 0; in < inner; ++in ) {
00335                             data_out.times.set( 0 );
00336
00337                             ( *alp_program )( data_in, data_out );
00338                             grb::collectives< implementation >::reduce( data_out.times.io, 0,
00339                                 grb::operators::max< double >() );
00340                             grb::collectives< implementation >::reduce( data_out.times.preamble, 0,
00341                                 grb::operators::max< double >() );
00342                             grb::collectives< implementation >::reduce( data_out.times.useful, 0,
00343                                 grb::operators::max< double >() );
00344                             grb::collectives< implementation >::reduce( data_out.times.postamble, 0,
00345                                 grb::operators::max< double >() );
00346                             inner_times.accum( data_out.times );
00347                         }
00348
00349                         // calculate performance stats
00350                         benchmark_calc_inner( out, inner, inner_times, total_times, min_times,
00351                             max_times, sdev_times );
00352
00353 #ifndef _GRB_NO_STDIO
00354                         // give experiment output line
00355                         if( pid == 0 ) {
00356                             std::cout « "Outer iteration #" « out « " timings "
00357                                 « "(io, preamble, useful, postamble, time since epoch): " « std::fixed
00358                                 « inner_times.io « ", " « inner_times.preamble « ", "
00359                                 « inner_times.useful « ", " « inner_times.postamble « ", ";
00360                                 printTimeSinceEpoch( false );
00361                             std::cout « std::scientific;
00362                         }
00363 #endif
00364
00365                         // pause for next outer loop
00366                         if( sleep( 1 ) != 0 ) {
00367 #ifndef _GRB_NO_STDIO
00368                             std::cerr « "Sleep interrupted, assume benchmark is unreliable; "
00369                                 « "exiting.\n";
00370 #endif
00371                             abort();
00372                         }
00373                     }
00374
00375                     // calculate performance stats
00376                     benchmark_calc_outer( outer, total_times, min_times, max_times, sdev_times,
00377                         pid );
00378                     delete[] sdev_times;
00379
00380                     return SUCCESS;
00381                 }
00382
00383
00384             public:
00385
00386                 BenchmarkerBase() {
00387 #ifndef _GRB_NO_STDIO
00388                     printTimeSinceEpoch();
00389 #endif
00390                 }
00391
00392         };
00393
00394     } // namespace internal
00395
```

```
00404     template< enum EXEC_MODE mode, enum Backend implementation >
00405     class Benchmarker {
00406
00407         public :
00408
00437             Benchmarker(
00438                 const size_t process_id = 0,
00439                 size_t nprocs = 1,
00440                 std::string hostname = "localhost",
00441                 std::string port = "0"
00442             ) {
00443                 (void)process_id; (void)nprocs; (void)hostname; (void)port;
00444 #ifndef _GRB_NO_EXCEPTIONS
00445                 throw std::logic_error( "Benchmarker class called with unsupported mode or "
00446                     "implementation" );
00447 #endif
00448             }
00449
00485             template< typename T, typename U >
00486             RC exec(
00487                 void ( *alp_program )( const T &, U & ),
00488                 const T &data_in,
00489                 U &data_out,
00490                 const size_t inner,
00491                 const size_t outer,
00492                 const bool broadcast = false
00493             ) const {
00494                 (void) alp_program;
00495                 (void) data_in;
00496                 (void) data_out;
00497                 (void) inner;
00498                 (void) outer;
00499                 (void) broadcast;
00500
00501                 // stub implementation, should be overridden by specialised implementation.
00502                 // furthermore, it should be impossible to call this function without
00503                 // triggering an exception during construction of this stub class, so we
00504                 // just return PANIC here
00505                 return PANIC;
00506             }
00507
00545             template< typename U >
00546             RC exec(
00547                 void ( *alp_program )( const void *, const size_t, U & ),
00548                 const void * data_in, const size_t in_size,
00549                 U &data_out,
00550                 const size_t inner, const size_t outer,
00551                 const bool broadcast = false
00552             ) const {
00553                 (void) alp_program;
00554                 (void) data_in;
00555                 (void) in_size;
00556                 (void) data_out;
00557                 (void) inner;
00558                 (void) outer;
00559                 (void) broadcast;
00560
00561                 // stub implementation, should be overridden by specialised implementation.
00562                 // furthermore, it should be impossible to call this function without
00563                 // triggering an exception during construction of this stub class, so we
00564                 // just return PANIC here
00565                 return PANIC;
00566             }
00567
00591             static RC finalize() {
00592                 return Launcher< mode, implementation >::finalize();
00593             }
00594
00595     };
00596
00597 } // end namespace “grb”
00598
00599 #endif // end _H_GRB_BENCH_BASE
00600
```

10.33 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< 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<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 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 using any number of vectors of equal length, following the nonzero pattern of the given vector x.

- 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 Input↩ Type2 , 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 Input↩ Type2 , 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 Input↩ Type2 , 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< Input↩ Type2 >::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.

10.33.1 Detailed Description

Defines the ALP/GraphBLAS level-1 API.

Author

A. N. Yzelman

Date

5th of December 2016

10.34 blas1.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_BASE_BLAS1
00028 #define _H_GRB_BASE_BLAS1
00029
00030 #include <graphblas/rc.hpp>
00031 #include <graphblas/ops.hpp>
00032 #include <graphblas/phase.hpp>
00033 #include <graphblas/monoid.hpp>
00034 #include <graphblas/backends.hpp>
00035 #include <graphblas/semiring.hpp>
00036 #include <graphblas/descriptors.hpp>
00037 #include <graphblas/internalops.hpp>
00038
```

```
00039 #include <assert.h>
00040
00041
00042 namespace grb {
00043
00154     #define NO_MASK Vector< bool >( 0 )
00155
00200     template<
00201         Descriptor descr = descriptors::no_operation,
00202         class OP, enum Backend backend,
00203         typename OutputType, typename InputType1, typename InputType2,
00204         typename Coords
00205     >
00206     RC eWiseApply(
00207         Vector< OutputType, backend, Coords > &z,
00208         const InputType1 alpha,
00209         const InputType2 beta,
00210         const OP &op = OP(),
00211         const Phase &phase = EXECUTE,
00212         const typename std::enable_if<
00213             !grb::is_object< OutputType >::value &&
00214             !grb::is_object< InputType1 >::value &&
00215             !grb::is_object< InputType2 >::value &&
00216             grb::is_operator< OP >::value, void
00217         >::type * const = nullptr
00218     ) {
00219 #ifdef _DEBUG
00220         std::cout « "In eWiseApply ([T1]<-T2<-T3), operator, base\n";
00221 #endif
00222 #ifndef NDEBUG
00223         const bool should_not_call_eWiseApplyOpASS_base = false;
00224         assert( should_not_call_eWiseApplyOpASS_base );
00225 #endif
00226         (void) z;
00227         (void) alpha;
00228         (void) beta;
00229         (void) op;
00230         (void) phase;
00231         return UNSUPPORTED;
00232     }
00233
00278     template<
00279         Descriptor descr = descriptors::no_operation,
00280         class Monoid, enum Backend backend,
00281         typename OutputType, typename InputType1, typename InputType2,
00282         typename Coords
00283     >
00284     RC eWiseApply(
00285         Vector< OutputType, backend, Coords > &z,
00286         const InputType1 alpha,
00287         const InputType2 beta,
00288         const Monoid &monoid = Monoid(),
00289         const Phase &phase = EXECUTE,
00290         const typename std::enable_if<
00291             !grb::is_object< OutputType >::value &&
00292             !grb::is_object< InputType1 >::value &&
00293             !grb::is_object< InputType2 >::value &&
00294             grb::is_monoid< Monoid >::value, void
00295         >::type * const = nullptr
00296     ) {
00297 #ifdef _DEBUG
00298         std::cout « "In eWiseApply ([T1]<-T2<-T3), monoid, base\n";
00299 #endif
00300 #ifndef NDEBUG
00301         const bool should_not_call_eWiseApplyMonASS_base = false;
00302         assert( should_not_call_eWiseApplyMonASS_base );
00303 #endif
00304         (void) z;
00305         (void) alpha;
00306         (void) beta;
00307         (void) monoid;
00308         (void) phase;
00309         return UNSUPPORTED;
00310     }
00311
00387     template<
00388         Descriptor descr = descriptors::no_operation,
00389         class OP, enum Backend backend,
00390         typename OutputType, typename InputType1, typename InputType2,
00391         typename Coords
00392     >
00393     RC eWiseApply(
00394         Vector< OutputType, backend, Coords > &z,
00395         const InputType1 alpha,
00396         const Vector< InputType2, backend, Coords > &y,
00397         const OP &op = OP(),
00398         const Phase &phase = EXECUTE,
```

```
00399         const typename std::enable_if<
00400             !grb::is_object< OutputType >::value &&
00401             !grb::is_object< InputType1 >::value &&
00402             !grb::is_object< InputType2 >::value &&
00403             grb::is_operator< OP >::value, void
00404         >::type * const = nullptr
00405     ) {
00406 #ifdef _DEBUG
00407         std::cout « "In eWiseApply ([T1]<-T2<-[T3]), operator, base\n";
00408 #endif
00409 #ifndef NDEBUG
00410         const bool should_not_call_eWiseApplyOpASA_base = false;
00411         assert( should_not_call_eWiseApplyOpASA_base );
00412 #endif
00413         (void) z;
00414         (void) alpha;
00415         (void) y;
00416         (void) op;
00417         (void) phase;
00418         return UNSUPPORTED;
00419     }
00420
00496     template<
00497         Descriptor descr = descriptors::no_operation,
00498         class OP, enum Backend backend,
00499         typename OutputType, typename MaskType,
00500         typename InputType1, typename InputType2,
00501         typename Coords
00502     >
00503     RC eWiseApply(
00504         Vector< OutputType, backend, Coords > &z,
00505         const Vector< MaskType, backend, Coords > &mask,
00506         const InputType1 alpha,
00507         const Vector< InputType2, backend, Coords > &y,
00508         const OP &op = OP(),
00509         const Phase &phase = EXECUTE,
00510         const typename std::enable_if<
00511             !grb::is_object< OutputType >::value &&
00512             !grb::is_object< MaskType >::value &&
00513             !grb::is_object< InputType1 >::value &&
00514             !grb::is_object< InputType2 >::value &&
00515             grb::is_operator< OP >::value, void
00516         >::type * const = nullptr
00517     ) {
00518 #ifdef _DEBUG
00519         std::cout « "In masked eWiseApply ([T1]<-T2<-[T3], operator, base)\n";
00520 #endif
00521 #ifndef NDEBUG
00522         const bool should_not_call_eWiseApplyOpAMSA_base = false;
00523         assert( should_not_call_eWiseApplyOpAMSA_base );
00524 #endif
00525         (void) z;
00526         (void) mask;
00527         (void) alpha;
00528         (void) y;
00529         (void) op;
00530         (void) phase;
00531         return UNSUPPORTED;
00532     }
00533
00608     template<
00609         Descriptor descr = descriptors::no_operation,
00610         class Monoid, enum Backend backend,
00611         typename OutputType, typename InputType1, typename InputType2,
00612         typename Coords
00613     >
00614     RC eWiseApply(
00615         Vector< OutputType, backend, Coords > &z,
00616         const InputType1 alpha,
00617         const Vector< InputType2, backend, Coords > &y,
00618         const Monoid &monoid = Monoid(),
00619         const Phase &phase = EXECUTE,
00620         const typename std::enable_if< !grb::is_object< OutputType >::value &&
00621             !grb::is_object< InputType1 >::value &&
00622             !grb::is_object< InputType2 >::value &&
00623             grb::is_monoid< Monoid >::value, void
00624         >::type * const = nullptr
00625     ) {
00626 #ifdef _DEBUG
00627         std::cout « "In unmasked eWiseApply ([T1]<-T2<-[T3], monoid, base)\n";
00628 #endif
00629 #ifndef NDEBUG
00630         const bool should_not_call_eWiseApplyMonoidASA_base = false;
00631         assert( should_not_call_eWiseApplyMonoidASA_base );
00632 #endif
00633         (void) z;
00634         (void) alpha;
```

```
00635         (void) y;
00636         (void) monoid;
00637         (void) phase;
00638         return UNSUPPORTED;
00639     }
00640
00719     template<
00720         Descriptor descr = descriptors::no_operation,
00721         class Monoid, enum Backend backend,
00722         typename OutputType, typename MaskType,
00723         typename InputType1, typename InputType2,
00724         typename Coords
00725     >
00726     RC eWiseApply(
00727         Vector< OutputType, backend, Coords > &z,
00728         const Vector< MaskType, backend, Coords > &mask,
00729         const InputType1 alpha,
00730         const Vector< InputType2, backend, Coords > &y,
00731         const Monoid &monoid = Monoid(),
00732         const Phase &phase = EXECUTE,
00733         const typename std::enable_if< !grb::is_object< OutputType >::value &&
00734             !grb::is_object< MaskType >::value &&
00735             !grb::is_object< InputType1 >::value &&
00736             !grb::is_object< InputType2 >::value &&
00737             grb::is_monoid< Monoid >::value,
00738         void >::type * const = nullptr
00739     ) {
00740 #ifdef _DEBUG
00741         std::cout << "In masked eWiseApply ([T1]<-T2<-[T3], using monoid)\n";
00742 #endif
00743 #ifndef NDEBUG
00744         const bool should_not_call_eWiseApplyMonoidAMSA_base = false;
00745         assert( should_not_call_eWiseApplyMonoidAMSA_base );
00746 #endif
00747         (void) z;
00748         (void) mask;
00749         (void) alpha;
00750         (void) y;
00751         (void) monoid;
00752         (void) phase;
00753         return UNSUPPORTED;
00754     }
00755
00832     template<
00833         Descriptor descr = descriptors::no_operation,
00834         class OP, enum Backend backend,
00835         typename OutputType, typename InputType1, typename InputType2,
00836         typename Coords
00837     >
00838     RC eWiseApply(
00839         Vector< OutputType, backend, Coords > &z,
00840         const Vector< InputType1, backend, Coords > &x,
00841         const InputType2 beta,
00842         const OP &op = OP(),
00843         const Phase &phase = EXECUTE,
00844         const typename std::enable_if<
00845             !grb::is_object< OutputType >::value &&
00846             !grb::is_object< InputType1 >::value &&
00847             !grb::is_object< InputType2 >::value &&
00848             grb::is_operator< OP >::value, void
00849         >::type * const = nullptr
00850     ) {
00851 #ifdef _DEBUG
00852         std::cout << "In eWiseApply ([T1]<-[T2]<-T3), operator, base\n";
00853 #endif
00854 #ifndef NDEBUG
00855         const bool should_not_call_eWiseApplyOpAAS_base = false;
00856         assert( should_not_call_eWiseApplyOpAAS_base );
00857 #endif
00858         (void) z;
00859         (void) x;
00860         (void) beta;
00861         (void) op;
00862         (void) phase;
00863         return UNSUPPORTED;
00864     }
00865
00943     template<
00944         Descriptor descr = descriptors::no_operation,
00945         class OP, enum Backend backend,
00946         typename OutputType, typename MaskType,
00947         typename InputType1, typename InputType2,
00948         typename Coords
00949     >
00950     RC eWiseApply(
00951         Vector< OutputType, backend, Coords > &z,
00952         const Vector< MaskType, backend, Coords > &mask,
```

```
00953         const Vector< InputType1, backend, Coords > &x,
00954         const InputType2 beta,
00955         const OP &op = OP(),
00956         const Phase &phase = EXECUTE,
00957         const typename std::enable_if< !grb::is_object< OutputType >::value &&
00958             !grb::is_object< MaskType >::value &&
00959             !grb::is_object< InputType1 >::value &&
00960             !grb::is_object< InputType2 >::value &&
00961             grb::is_operator< OP >::value, void
00962         >::type * const = nullptr
00963     ) {
00964 #ifdef _DEBUG
00965         std::cout « "In masked eWiseApply ([T1]<-[T2]<-T3, operator, base)\n";
00966 #endif
00967 #ifndef NDEBUG
00968         const bool should_not_call_eWiseApplyOpAMAS_base = false;
00969         assert( should_not_call_eWiseApplyOpAMAS_base );
00970 #endif
00971         (void) z;
00972         (void) mask;
00973         (void) x;
00974         (void) beta;
00975         (void) op;
00976         (void) phase;
00977         return UNSUPPORTED;
00978     }
00979
01054     template<
01055         Descriptor descr = descriptors::no_operation,
01056         class Monoid, enum Backend backend,
01057         typename OutputType, typename InputType1, typename InputType2,
01058         typename Coords
01059     >
01060     RC eWiseApply(
01061         Vector< OutputType, backend, Coords > &z,
01062         const Vector< InputType1, backend, Coords > &x,
01063         const InputType2 beta,
01064         const Monoid &monoid = Monoid(),
01065         const Phase &phase = EXECUTE,
01066         const typename std::enable_if< !grb::is_object< OutputType >::value &&
01067                 !grb::is_object< InputType1 >::value &&
01068                 !grb::is_object< InputType2 >::value &&
01069                 grb::is_monoid< Monoid >::value,
01070             void >::type * const = nullptr
01071     ) {
01072 #ifdef _DEBUG
01073         std::cout « "In unmasked eWiseApply ([T1]<-[T2]<-T3, monoid, base)\n";
01074 #endif
01075 #ifndef NDEBUG
01076         const bool should_not_call_eWiseApplyMonoidAAS_base = false;
01077         assert( should_not_call_eWiseApplyMonoidAAS_base );
01078 #endif
01079         (void) z;
01080         (void) x;
01081         (void) beta;
01082         (void) monoid;
01083         (void) phase;
01084         return UNSUPPORTED;
01085     }
01086
01165     template<
01166         Descriptor descr = descriptors::no_operation,
01167         class Monoid, enum Backend backend,
01168         typename OutputType, typename MaskType,
01169         typename InputType1, typename InputType2,
01170         typename Coords
01171     >
01172     RC eWiseApply(
01173         Vector< OutputType, backend, Coords > &z,
01174         const Vector< MaskType, backend, Coords > &mask,
01175         const Vector< InputType1, backend, Coords > &x,
01176         const InputType2 beta,
01177         const Monoid &monoid = Monoid(),
01178         const Phase &phase = EXECUTE,
01179         const typename std::enable_if< !grb::is_object< OutputType >::value &&
01180             !grb::is_object< MaskType >::value &&
01181             !grb::is_object< InputType1 >::value &&
01182             !grb::is_object< InputType2 >::value &&
01183             grb::is_monoid< Monoid >::value, void
01184         >::type * const = nullptr
01185     ) {
01186 #ifdef _DEBUG
01187         std::cout « "In masked eWiseApply ([T1]<-[T2]<-T3, monoid, base)\n";
01188 #endif
01189 #ifndef NDEBUG
01190         const bool should_not_call_eWiseApplyMonoidAMAS_base = false;
01191         assert( should_not_call_eWiseApplyMonoidAMAS_base );
```

```
01192 #endif
01193         (void) z;
01194         (void) mask;
01195         (void) x;
01196         (void) beta;
01197         (void) monoid;
01198         (void) phase;
01199         return UNSUPPORTED;
01200     }
01201
01278     template<
01279         Descriptor descr = descriptors::no_operation,
01280         class OP, enum Backend backend,
01281         typename OutputType, typename InputType1, typename InputType2,
01282         typename Coords
01283     >
01284     RC eWiseApply(
01285         Vector< OutputType, backend, Coords > &z,
01286         const Vector< InputType1, backend, Coords > &x,
01287         const Vector< InputType2, backend, Coords > &y,
01288         const OP &op = OP(),
01289         const Phase &phase = EXECUTE,
01290         const typename std::enable_if< !grb::is_object< OutputType >::value &&
01291             !grb::is_object< InputType1 >::value &&
01292             !grb::is_object< InputType2 >::value &&
01293             grb::is_operator< OP >::value, void
01294         >::type * const = nullptr
01295     ) {
01296 #ifdef _DEBUG
01297         std::cout « "In eWiseApply ([T1]<-[T2]<-[T3]), operator variant\n";
01298 #endif
01299 #ifndef NDEBUG
01300         const bool should_not_call_eWiseApplyOpAAA_base = false;
01301         assert( should_not_call_eWiseApplyOpAAA_base );
01302 #endif
01303         (void) z;
01304         (void) x;
01305         (void) y;
01306         (void) op;
01307         (void) phase;
01308         return UNSUPPORTED;
01309     }
01310
01390     template<
01391         Descriptor descr = descriptors::no_operation,
01392         class OP, enum Backend backend,
01393         typename OutputType, typename MaskType,
01394         typename InputType1, typename InputType2,
01395         typename Coords
01396     >
01397     RC eWiseApply(
01398         Vector< OutputType, backend, Coords > &z,
01399         const Vector< MaskType, backend, Coords > &mask,
01400         const Vector< InputType1, backend, Coords > &x,
01401         const Vector< InputType2, backend, Coords > &y,
01402         const OP &op = OP(),
01403         const Phase &phase = EXECUTE,
01404         const typename std::enable_if<
01405             !grb::is_object< OutputType >::value &&
01406             !grb::is_object< MaskType >::value &&
01407             !grb::is_object< InputType1 >::value &&
01408             !grb::is_object< InputType2 >::value &&
01409             grb::is_operator< OP >::value, void
01410         >::type * const = nullptr
01411     ) {
01412 #ifdef _DEBUG
01413         std::cout « "In masked eWiseApply ([T1]<-[T2]<-[T3], operator, base)\n";
01414 #endif
01415 #ifndef NDEBUG
01416         const bool should_not_call_eWiseApplyOpAMAA_base = false;
01417         assert( should_not_call_eWiseApplyOpAMAA_base );
01418 #endif
01419         (void) z;
01420         (void) mask;
01421         (void) x;
01422         (void) y;
01423         (void) op;
01424         (void) phase;
01425         return UNSUPPORTED;
01426     }
01427
01505     template<
01506         Descriptor descr = descriptors::no_operation,
01507         class Monoid, enum Backend backend,
01508         typename OutputType, typename InputType1, typename InputType2,
01509         typename Coords
01510     >
```

```
01511     RC eWiseApply(
01512         Vector< OutputType, backend, Coords > &z,
01513         const Vector< InputType1, backend, Coords > &x,
01514         const Vector< InputType2, backend, Coords > &y,
01515         const Monoid &monoid = Monoid(),
01516         const Phase &phase = EXECUTE,
01517         const typename std::enable_if< !grb::is_object< OutputType >::value &&
01518             !grb::is_object< InputType1 >::value &&
01519             !grb::is_object< InputType2 >::value &&
01520             grb::is_monoid< Monoid >::value, void
01521         >::type * const = nullptr
01522     ) {
01523 #ifdef _DEBUG
01524         std::cout « "In unmasked eWiseApply ([T1]<-[T2]<-[T3], monoid, base)\n";
01525 #endif
01526 #ifndef NDEBUG
01527         const bool should_not_call_eWiseApplyOpAMAA_base = false;
01528         assert( should_not_call_eWiseApplyOpAMAA_base );
01529 #endif
01530         (void) z;
01531         (void) x;
01532         (void) y;
01533         (void) monoid;
01534         (void) phase;
01535         return UNSUPPORTED;
01536     }
01537
01613     template<
01614         Descriptor descr = descriptors::no_operation,
01615         class Monoid, enum Backend backend,
01616         typename OutputType, typename MaskType,
01617         typename InputType1, typename InputType2,
01618         typename Coords
01619     >
01620     RC eWiseApply(
01621         Vector< OutputType, backend, Coords > &z,
01622         const Vector< MaskType, backend, Coords > &mask,
01623         const Vector< InputType1, backend, Coords > &x,
01624         const Vector< InputType2, backend, Coords > &y,
01625         const Monoid &monoid = Monoid(),
01626         const Phase &phase = EXECUTE,
01627         const typename std::enable_if<
01628             !grb::is_object< OutputType >::value &&
01629             !grb::is_object< MaskType >::value &&
01630             !grb::is_object< InputType1 >::value &&
01631             !grb::is_object< InputType2 >::value &&
01632             grb::is_monoid< Monoid >::value, void
01633         >::type * const = nullptr
01634     ) {
01635 #ifdef _DEBUG
01636         std::cout « "In masked eWiseApply ([T1]<-[T2]<-[T3], monoid, base)\n";
01637 #endif
01638 #ifndef NDEBUG
01639         const bool should_not_call_eWiseApplyMonoidAMAA_base = false;
01640         assert( should_not_call_eWiseApplyMonoidAMAA_base );
01641 #endif
01642         (void) z;
01643         (void) mask;
01644         (void) x;
01645         (void) y;
01646         (void) monoid;
01647         (void) phase;
01648         return UNSUPPORTED;
01649     }
01650
01730     template<
01731         Descriptor descr = descriptors::no_operation,
01732         class Ring, enum Backend backend,
01733         typename OutputType, typename InputType1, typename InputType2,
01734         typename Coords
01735     >
01736     RC eWiseAdd(
01737         Vector< OutputType, backend, Coords > &z,
01738         const Vector< InputType1, backend, Coords > &x,
01739         const Vector< InputType2, backend, Coords > &y,
01740         const Ring &ring = Ring(),
01741         const Phase &phase = EXECUTE,
01742         const typename std::enable_if< !grb::is_object< OutputType >::value &&
01743             !grb::is_object< InputType1 >::value &&
01744             !grb::is_object< InputType2 >::value &&
01745             grb::is_semiring< Ring >::value, void
01746         >::type * const = nullptr
01747     ) {
01748 #ifdef _DEBUG
01749         std::cout « "in eWiseAdd ([T1] <- [T2] + [T3]), unmasked, base";
01750 #endif
01751 #ifndef NDEBUG
```

```
01752         const bool should_not_call_eWiseAddAAA_base = false;
01753         assert( should_not_call_eWiseAddAAA_base );
01754 #endif
01755         (void) z;
01756         (void) x;
01757         (void) y;
01758         (void) ring;
01759         (void) phase;
01760         return UNSUPPORTED;
01761     }
01762
01837     template<
01838         Descriptor descr = descriptors::no_operation,
01839         class Ring, enum Backend backend,
01840         typename InputType1, typename InputType2, typename OutputType,
01841         typename Coords
01842     >
01843     RC eWiseAdd(
01844         Vector< OutputType, backend, Coords > &z,
01845         const InputType1 alpha,
01846         const Vector< InputType2, backend, Coords > &y,
01847         const Ring &ring = Ring(),
01848         const Phase &phase = EXECUTE,
01849         const typename std::enable_if< !grb::is_object< OutputType >::value &&
01850             !grb::is_object< InputType1 >::value &&
01851             !grb::is_object< InputType2 >::value &&
01852             grb::is_semiring< Ring >::value, void
01853         >::type * const = nullptr
01854     ) {
01855 #ifdef _DEBUG
01856         std::cout « "in eWiseAdd ([T1] <- T2 + [T3]), unmasked, base";
01857 #endif
01858 #ifndef NDEBUG
01859         const bool should_not_call_eWiseAddASA_base = false;
01860         assert( should_not_call_eWiseAddASA_base );
01861 #endif
01862         (void) z;
01863         (void) alpha;
01864         (void) y;
01865         (void) ring;
01866         (void) phase;
01867         return UNSUPPORTED;
01868     }
01869
01944     template<
01945         Descriptor descr = descriptors::no_operation,
01946         class Ring, enum Backend backend,
01947         typename InputType1, typename InputType2, typename OutputType,
01948         typename Coords
01949     >
01950     RC eWiseAdd(
01951         Vector< OutputType, backend, Coords > &z,
01952         const Vector< InputType1, backend, Coords > &x,
01953         const InputType2 beta,
01954         const Ring &ring = Ring(),
01955         const Phase &phase = EXECUTE,
01956         const typename std::enable_if< !grb::is_object< OutputType >::value &&
01957             !grb::is_object< InputType1 >::value &&
01958             !grb::is_object< InputType2 >::value &&
01959             grb::is_semiring< Ring >::value, void
01960         >::type * const = nullptr
01961     ) {
01962 #ifdef _DEBUG
01963         std::cout « "in eWiseAdd ([T1] <- [T2] + T3), unmasked, base";
01964 #endif
01965 #ifndef NDEBUG
01966         const bool should_not_call_eWiseAddAAS_base = false;
01967         assert( should_not_call_eWiseAddAAS_base );
01968 #endif
01969         (void) z;
01970         (void) x;
01971         (void) beta;
01972         (void) ring;
01973         (void) phase;
01974         return UNSUPPORTED;
01975     }
01976
02047     template<
02048         Descriptor descr = descriptors::no_operation,
02049         class Ring, enum Backend backend,
02050         typename InputType1, typename InputType2, typename OutputType,
02051         typename Coords
02052     >
02053     RC eWiseAdd(
02054         Vector< OutputType, backend, Coords > &z,
02055         const InputType1 alpha,
02056         const InputType2 beta,
```

```
02057         const Ring &ring = Ring(),
02058         const Phase &phase = EXECUTE,
02059         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02060             !grb::is_object< InputType1 >::value &&
02061             !grb::is_object< InputType2 >::value &&
02062             grb::is_semiring< Ring >::value, void
02063         >::type * const = nullptr
02064     ) {
02065 #ifdef _DEBUG
02066         std::cout << "in eWiseAdd ([T1] <- T2 + T3), unmasked, base";
02067 #endif
02068 #ifndef NDEBUG
02069         const bool should_not_call_eWiseAddASS_base = false;
02070         assert( should_not_call_eWiseAddASS_base );
02071 #endif
02072         (void) z;
02073         (void) alpha;
02074         (void) beta;
02075         (void) ring;
02076         (void) phase;
02077         return UNSUPPORTED;
02078     }
02079
02165     template<
02166         Descriptor descr = descriptors::no_operation,
02167         class Ring, enum Backend backend,
02168         typename OutputType, typename MaskType,
02169         typename InputType1, typename InputType2,
02170         typename Coords
02171     >
02172     RC eWiseAdd(
02173         Vector< OutputType, backend, Coords > &z,
02174         const Vector< MaskType, backend, Coords > &mask,
02175         const Vector< InputType1, backend, Coords > &x,
02176         const Vector< InputType2, backend, Coords > &y,
02177         const Ring &ring = Ring(),
02178         const Phase &phase = EXECUTE,
02179         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02180             !grb::is_object< InputType1 >::value &&
02181             !grb::is_object< InputType2 >::value &&
02182             grb::is_semiring< Ring >::value, void
02183         >::type * const = nullptr
02184     ) {
02185 #ifdef _DEBUG
02186         std::cout << "in eWiseAdd ([T1] <- [T2] + [T3]), masked, base";
02187 #endif
02188 #ifndef NDEBUG
02189         const bool should_not_call_eWiseAddAMAA_base = false;
02190         assert( should_not_call_eWiseAddAMAA_base );
02191 #endif
02192         (void) z;
02193         (void) mask;
02194         (void) x;
02195         (void) y;
02196         (void) ring;
02197         (void) phase;
02198         return UNSUPPORTED;
02199     }
02200
02282     template<
02283         Descriptor descr = descriptors::no_operation,
02284         class Ring, enum Backend backend,
02285         typename InputType1, typename InputType2,
02286         typename OutputType, typename MaskType,
02287         typename Coords
02288     >
02289     RC eWiseAdd(
02290         Vector< OutputType, backend, Coords > &z,
02291         const Vector< MaskType, backend, Coords > &mask,
02292         const InputType1 alpha,
02293         const Vector< InputType2, backend, Coords > &y,
02294         const Ring &ring = Ring(),
02295         const Phase &phase = EXECUTE,
02296         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02297             !grb::is_object< InputType1 >::value &&
02298             !grb::is_object< InputType2 >::value &&
02299             grb::is_semiring< Ring >::value, void
02300         >::type * const = nullptr
02301     ) {
02302 #ifdef _DEBUG
02303         std::cout << "in eWiseAdd ([T1] <- T2 + [T3]), masked, base";
02304 #endif
02305 #ifndef NDEBUG
02306         const bool should_not_call_eWiseAddAMSA_base = false;
02307         assert( should_not_call_eWiseAddAMSA_base );
02308 #endif
02309         (void) z;
```

```
02310         (void) mask;
02311         (void) alpha;
02312         (void) y;
02313         (void) ring;
02314         (void) phase;
02315         return UNSUPPORTED;
02316     }
02317
02399     template<
02400         Descriptor descr = descriptors::no_operation,
02401         class Ring, enum Backend backend,
02402         typename InputType1, typename InputType2,
02403         typename OutputType, typename MaskType,
02404         typename Coords
02405     >
02406     RC eWiseAdd(
02407         Vector< OutputType, backend, Coords > &z,
02408         const Vector< MaskType, backend, Coords > &mask,
02409         const Vector< InputType1, backend, Coords > &x,
02410         const InputType2 beta,
02411         const Ring &ring = Ring(),
02412         const Phase &phase = EXECUTE,
02413         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02414             !grb::is_object< InputType1 >::value &&
02415             !grb::is_object< InputType2 >::value &&
02416             grb::is_semiring< Ring >::value, void
02417         >::type * const = nullptr
02418     ) {
02419 #ifdef _DEBUG
02420         std::cout « "in eWiseAdd ([T1] <- [T2] + T3), masked, base";
02421 #endif
02422 #ifndef NDEBUG
02423         const bool should_not_call_eWiseAddAMAS_base = false;
02424         assert( should_not_call_eWiseAddAMAS_base );
02425 #endif
02426         (void) z;
02427         (void) mask;
02428         (void) x;
02429         (void) beta;
02430         (void) ring;
02431         (void) phase;
02432         return UNSUPPORTED;
02433     }
02434
02512     template<
02513         Descriptor descr = descriptors::no_operation,
02514         class Ring, enum Backend backend,
02515         typename InputType1, typename InputType2,
02516         typename OutputType, typename MaskType,
02517         typename Coords
02518     >
02519     RC eWiseAdd(
02520         Vector< OutputType, backend, Coords > &z,
02521         const Vector< MaskType, backend, Coords > &mask,
02522         const InputType1 alpha,
02523         const InputType2 beta,
02524         const Ring &ring = Ring(),
02525         const Phase &phase = EXECUTE,
02526         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02527             !grb::is_object< InputType1 >::value &&
02528             !grb::is_object< InputType2 >::value &&
02529             grb::is_semiring< Ring >::value, void
02530         >::type * const = nullptr
02531     ) {
02532 #ifdef _DEBUG
02533         std::cout « "in eWiseAdd ([T1] <- T2 + T3), masked, base";
02534 #endif
02535 #ifndef NDEBUG
02536         const bool should_not_call_eWiseAddAMSS_base = false;
02537         assert( should_not_call_eWiseAddAMSS_base );
02538 #endif
02539         (void) z;
02540         (void) mask;
02541         (void) alpha;
02542         (void) beta;
02543         (void) ring;
02544         (void) phase;
02545         return UNSUPPORTED;
02546     }
02547
02612     template<
02613         Descriptor descr = descriptors::no_operation,
02614         class Ring, enum Backend backend,
02615         typename InputType1, typename InputType2, typename OutputType,
02616         typename Coords
02617     >
02618     RC eWiseMul(
```

```
02619         Vector< OutputType, backend, Coords > &z,
02620         const Vector< InputType1, backend, Coords > &x,
02621         const Vector< InputType2, backend, Coords > &y,
02622         const Ring &ring = Ring(),
02623         const Phase &phase = EXECUTE,
02624         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02625             !grb::is_object< InputType1 >::value &&
02626             !grb::is_object< InputType2 >::value &&
02627             grb::is_semiring< Ring >::value, void
02628         >::type * const = nullptr
02629     ) {
02630 #ifdef _DEBUG
02631         std::cout « "in eWiseMul ([T1] <- [T2] * [T3]), unmasked, base";
02632 #endif
02633 #ifndef NDEBUG
02634         const bool should_not_call_eWiseMulAAA_base = false;
02635         assert( should_not_call_eWiseMulAAA_base );
02636 #endif
02637         (void) z;
02638         (void) x;
02639         (void) y;
02640         (void) ring;
02641         (void) phase;
02642         return UNSUPPORTED;
02643     }
02644
02709     template<
02710         Descriptor descr = descriptors::no_operation,
02711         class Ring, enum Backend backend,
02712         typename InputType1, typename InputType2, typename OutputType,
02713         typename Coords
02714     >
02715     RC eWiseMul(
02716         Vector< OutputType, backend, Coords > &z,
02717         const InputType1 alpha,
02718         const Vector< InputType2, backend, Coords > &y,
02719         const Ring &ring = Ring(),
02720         const Phase &phase = EXECUTE,
02721         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02722             !grb::is_object< InputType1 >::value &&
02723             !grb::is_object< InputType2 >::value &&
02724             grb::is_semiring< Ring >::value, void
02725         >::type * const = nullptr
02726     ) {
02727 #ifdef _DEBUG
02728         std::cout « "in eWiseMul ([T1] <- T2 * [T3]), unmasked, base";
02729 #endif
02730 #ifndef NDEBUG
02731         const bool should_not_call_eWiseMulASA_base = false;
02732         assert( should_not_call_eWiseMulASA_base );
02733 #endif
02734         (void) z;
02735         (void) alpha;
02736         (void) y;
02737         (void) ring;
02738         (void) phase;
02739         return UNSUPPORTED;
02740     }
02741
02806     template<
02807         Descriptor descr = descriptors::no_operation,
02808         class Ring, enum Backend backend,
02809         typename InputType1, typename InputType2, typename OutputType,
02810         typename Coords
02811     >
02812     RC eWiseMul(
02813         Vector< OutputType, backend, Coords > &z,
02814         const Vector< InputType1, backend, Coords > &x,
02815         const InputType2 beta,
02816         const Ring &ring = Ring(),
02817         const Phase &phase = EXECUTE,
02818         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02819             !grb::is_object< InputType1 >::value &&
02820             !grb::is_object< InputType2 >::value &&
02821             grb::is_semiring< Ring >::value, void
02822         >::type * const = nullptr
02823     ) {
02824 #ifdef _DEBUG
02825         std::cout « "in eWiseMul ([T1] <- [T2] * T3), unmasked, base";
02826 #endif
02827 #ifndef NDEBUG
02828         const bool should_not_call_eWiseMulAAS_base = false;
02829         assert( should_not_call_eWiseMulAAS_base );
02830 #endif
02831         (void) z;
02832         (void) x;
02833         (void) beta;
```

```
02834         (void) ring;
02835         (void) phase;
02836         return UNSUPPORTED;
02837     }
02838
02899     template<
02900         Descriptor descr = descriptors::no_operation,
02901         class Ring, enum Backend backend,
02902         typename InputType1, typename InputType2, typename OutputType,
02903         typename Coords
02904     >
02905     RC eWiseMul(
02906         Vector< OutputType, backend, Coords > &z,
02907         const InputType1 alpha,
02908         const InputType2 beta,
02909         const Ring &ring = Ring(),
02910         const Phase &phase = EXECUTE,
02911         const typename std::enable_if< !grb::is_object< OutputType >::value &&
02912             !grb::is_object< InputType1 >::value &&
02913             !grb::is_object< InputType2 >::value &&
02914             grb::is_semiring< Ring >::value, void
02915         >::type * const = nullptr
02916     ) {
02917 #ifdef _DEBUG
02918         std::cout << "in eWiseMul ([T1] <- T2 * T3), unmasked, base";
02919 #endif
02920 #ifndef NDEBUG
02921         const bool should_not_call_eWiseMulASS_base = false;
02922         assert( should_not_call_eWiseMulASS_base );
02923 #endif
02924         (void) z;
02925         (void) alpha;
02926         (void) beta;
02927         (void) ring;
02928         (void) phase;
02929         return UNSUPPORTED;
02930     }
02931
03001     template<
03002         Descriptor descr = descriptors::no_operation,
03003         class Ring, enum Backend backend,
03004         typename InputType1, typename InputType2,
03005         typename OutputType, typename MaskType,
03006         typename Coords
03007     >
03008     RC eWiseMul(
03009         Vector< OutputType, backend, Coords > &z,
03010         const Vector< MaskType, backend, Coords > &mask,
03011         const Vector< InputType1, backend, Coords > &x,
03012         const Vector< InputType2, backend, Coords > &y,
03013         const Ring &ring = Ring(),
03014         const Phase &phase = EXECUTE,
03015         const typename std::enable_if< !grb::is_object< OutputType >::value &&
03016             !grb::is_object< InputType1 >::value &&
03017             !grb::is_object< InputType2 >::value &&
03018             grb::is_semiring< Ring >::value, void
03019         >::type * const = nullptr
03020     ) {
03021 #ifdef _DEBUG
03022         std::cout << "in eWiseMul ([T1] <- [T2] * [T3]), masked, base";
03023 #endif
03024 #ifndef NDEBUG
03025         const bool should_not_call_eWiseMulAMAA_base = false;
03026         assert( should_not_call_eWiseMulAMAA_base );
03027 #endif
03028         (void) z;
03029         (void) mask;
03030         (void) x;
03031         (void) y;
03032         (void) ring;
03033         (void) phase;
03034         return UNSUPPORTED;
03035     }
03036
03106     template<
03107         Descriptor descr = descriptors::no_operation,
03108         class Ring, enum Backend backend,
03109         typename InputType1, typename InputType2,
03110         typename OutputType, typename MaskType,
03111         typename Coords
03112     >
03113     RC eWiseMul(
03114         Vector< OutputType, backend, Coords > &z,
03115         const Vector< MaskType, backend, Coords > &mask,
03116         const InputType1 alpha,
03117         const Vector< InputType2, backend, Coords > &y,
03118         const Ring &ring = Ring(),
```

```
03119         const Phase &phase = EXECUTE,
03120         const typename std::enable_if< !grb::is_object< OutputType >::value &&
03121             !grb::is_object< InputType1 >::value &&
03122             !grb::is_object< InputType2 >::value &&
03123             grb::is_semiring< Ring >::value, void
03124         >::type * const = nullptr
03125     ) {
03126 #ifdef _DEBUG
03127         std::cout « "in eWiseMul ([T1] <- T2 * [T3]), masked, base";
03128 #endif
03129 #ifndef NDEBUG
03130         const bool should_not_call_eWiseMulAMSA_base = false;
03131         assert( should_not_call_eWiseMulAMSA_base );
03132 #endif
03133         (void) z;
03134         (void) mask;
03135         (void) alpha;
03136         (void) y;
03137         (void) ring;
03138         (void) phase;
03139         return UNSUPPORTED;
03140     }
03141
03211     template<
03212         Descriptor descr = descriptors::no_operation,
03213         class Ring, enum Backend backend,
03214         typename InputType1, typename InputType2,
03215         typename OutputType, typename MaskType,
03216         typename Coords
03217     >
03218     RC eWiseMul(
03219         Vector< OutputType, backend, Coords > &z,
03220         const Vector< MaskType, backend, Coords > &mask,
03221         const Vector< InputType1, backend, Coords > &x,
03222         const InputType2 beta,
03223         const Ring &ring = Ring(),
03224         const Phase &phase = EXECUTE,
03225         const typename std::enable_if< !grb::is_object< OutputType >::value &&
03226             !grb::is_object< InputType1 >::value &&
03227             !grb::is_object< InputType2 >::value &&
03228             grb::is_semiring< Ring >::value, void
03229         >::type * const = nullptr
03230     ) {
03231 #ifdef _DEBUG
03232         std::cout « "in eWiseMul ([T1] <- [T2] * T3), masked, base";
03233 #endif
03234 #ifndef NDEBUG
03235         const bool should_not_call_eWiseMulAMAS_base = false;
03236         assert( should_not_call_eWiseMulAMAS_base );
03237 #endif
03238         (void) z;
03239         (void) mask;
03240         (void) x;
03241         (void) beta;
03242         (void) ring;
03243         (void) phase;
03244         return UNSUPPORTED;
03245     }
03246
03313     template<
03314         Descriptor descr = descriptors::no_operation,
03315         class Ring, enum Backend backend,
03316         typename InputType1, typename InputType2,
03317         typename OutputType, typename MaskType,
03318         typename Coords
03319     >
03320     RC eWiseMul(
03321         Vector< OutputType, backend, Coords > &z,
03322         const Vector< MaskType, backend, Coords > &mask,
03323         const InputType1 alpha,
03324         const InputType2 beta,
03325         const Ring &ring = Ring(),
03326         const Phase &phase = EXECUTE,
03327         const typename std::enable_if< !grb::is_object< OutputType >::value &&
03328             !grb::is_object< InputType1 >::value &&
03329             !grb::is_object< InputType2 >::value &&
03330             grb::is_semiring< Ring >::value, void
03331         >::type * const = nullptr
03332     ) {
03333 #ifdef _DEBUG
03334         std::cout « "in eWiseMul ([T1] <- T2 * T3), masked, base";
03335 #endif
03336 #ifndef NDEBUG
03337         const bool should_not_call_eWiseMulAMSS_base = false;
03338         assert( should_not_call_eWiseMulAMSS_base );
03339 #endif
03340         (void) z;
```

```
03341         (void) mask;
03342         (void) alpha;
03343         (void) beta;
03344         (void) ring;
03345         (void) phase;
03346         return UNSUPPORTED;
03347     }
03348
03517     template<
03518         typename Func,
03519         typename DataType,
03520         Backend backend,
03521         typename Coords,
03522         typename... Args
03523     >
03524     RC eWiseLambda(
03525         const Func f,
03526         const Vector< DataType, backend, Coords > & x, Args...
03527     ) {
03528 #ifndef NDEBUG
03529         const bool should_not_call_base_vector_ewiselambda = false;
03530         assert( should_not_call_base_vector_ewiselambda );
03531 #endif
03532         (void) f;
03533         (void) x;
03534         return UNSUPPORTED;
03535     }
03536
03612     template<
03613         Descriptor descr = descriptors::no_operation,
03614         class Monoid,
03615         typename InputType, typename IOType, typename MaskType,
03616         Backend backend, typename Coords
03617     >
03618     RC foldl(
03619         IOType &x,
03620         const Vector< InputType, backend, Coords > &y,
03621         const Vector< MaskType, backend, Coords > &mask,
03622         const Monoid &monoid = Monoid(),
03623         const typename std::enable_if< !grb::is_object< IOType >::value &&
03624             !grb::is_object< InputType >::value &&
03625             !grb::is_object< MaskType >::value &&
03626             grb::is_monoid< Monoid >::value, void
03627         >::type * const = nullptr
03628     ) {
03629 #ifndef NDEBUG
03630         const bool should_not_call_base_scalar_foldl = false;
03631         assert( should_not_call_base_scalar_foldl );
03632 #endif
03633         (void) y;
03634         (void) x;
03635         (void) mask;
03636         (void) monoid;
03637         return UNSUPPORTED;
03638     }
03639
03645     template<
03646         Descriptor descr = descriptors::no_operation,
03647         class Monoid,
03648         typename IOType, typename InputType,
03649         Backend backend,
03650         typename Coords
03651     >
03652     RC foldl(
03653         IOType &x,
03654         const Vector< InputType, backend, Coords > &y,
03655         const Monoid &monoid = Monoid(),
03656         const typename std::enable_if<
03657             !grb::is_object< IOType >::value &&
03658             grb::is_monoid< Monoid >::value,
03659         void >::type * const = nullptr
03660     ) {
03661 #ifndef NDEBUG
03662         const bool should_not_call_base_scalar_foldl_nomask = false;
03663         assert( should_not_call_base_scalar_foldl_nomask );
03664 #endif
03665         (void) y;
03666         (void) x;
03667         (void) monoid;
03668         return UNSUPPORTED;
03669     }
03670
03681     template<
03682         Descriptor descr = descriptors::no_operation,
03683         class OP,
03684         typename IOType, typename InputType, typename MaskType,
03685         Backend backend, typename Coords
```

```
03686     >
03687     RC foldl(
03688         IOType &x,
03689         const Vector< InputType, backend, Coords > &y,
03690         const Vector< MaskType, backend, Coords > &mask,
03691         const OP &op = OP(),
03692         const typename std::enable_if<
03693             !grb::is_object< IOType >::value &&
03694             !grb::is_object< MaskType >::value &&
03695             grb::is_operator< OP >::value,
03696         void >::type * const = nullptr
03697     ) {
03698 #ifndef NDEBUG
03699         const bool should_not_call_base_scalar_foldl_op = false;
03700         assert( should_not_call_base_scalar_foldl_op );
03701 #endif
03702         (void) x;
03703         (void) y;
03704         (void) mask;
03705         (void) op;
03706         return UNSUPPORTED;
03707     }
03708
03715     template<
03716         Descriptor descr = descriptors::no_operation,
03717         class Monoid,
03718         typename InputType, typename IOType, typename MaskType,
03719         Backend backend, typename Coords
03720     >
03721     RC foldr(
03722         const Vector< InputType, backend, Coords > &x,
03723         const Vector< MaskType, backend, Coords > &mask,
03724         IOType &y,
03725         const Monoid &monoid = Monoid(),
03726         const typename std::enable_if< !grb::is_object< IOType >::value &&
03727             !grb::is_object< InputType >::value &&
03728             !grb::is_object< MaskType >::value &&
03729             grb::is_monoid< Monoid >::value, void
03730         >::type * const = nullptr
03731     ) {
03732 #ifndef NDEBUG
03733         const bool should_not_call_base_scalar_foldr = false;
03734         assert( should_not_call_base_scalar_foldr );
03735 #endif
03736         (void) y;
03737         (void) x;
03738         (void) mask;
03739         (void) monoid;
03740         return UNSUPPORTED;
03741     }
03742
03749     template<
03750         Descriptor descr = descriptors::no_operation,
03751         class Monoid,
03752         typename IOType, typename InputType,
03753         Backend backend, typename Coords
03754     >
03755     RC foldr(
03756         const Vector< InputType, backend, Coords > &y,
03757         IOType &x,
03758         const Monoid &monoid = Monoid(),
03759         const typename std::enable_if<
03760             !grb::is_object< IOType >::value &&
03761             grb::is_monoid< Monoid >::value,
03762         void >::type * const = nullptr
03763     ) {
03764 #ifndef NDEBUG
03765         const bool should_not_call_base_scalar_foldr_nomask = false;
03766         assert( should_not_call_base_scalar_foldr_nomask );
03767 #endif
03768         (void) y;
03769         (void) x;
03770         (void) monoid;
03771         return UNSUPPORTED;
03772     }
03773
03828     template<
03829         Descriptor descr = descriptors::no_operation,
03830         class AddMonoid, class AnyOp,
03831         typename OutputType, typename InputType1, typename InputType2,
03832         enum Backend backend, typename Coords
03833     >
03834     RC dot(
03835         OutputType &z,
03836         const Vector< InputType1, backend, Coords > &x,
03837         const Vector< InputType2, backend, Coords > &y,
03838         const AddMonoid &addMonoid = AddMonoid(),
```

```
03839         const AnyOp &anyOp = AnyOp(),
03840         const Phase &phase = EXECUTE,
03841         const typename std::enable_if< !grb::is_object< OutputType >::value &&
03842             !grb::is_object< InputType1 >::value &&
03843             !grb::is_object< InputType2 >::value &&
03844             grb::is_monoid< AddMonoid >::value &&
03845             grb::is_operator< AnyOp >::value,
03846         void >::type * const = nullptr
03847     ) {
03848 #ifdef _DEBUG
03849         std::cout << "Should not call base grb::dot (monoid-operator version)\n";
03850 #endif
03851 #ifndef NDEBUG
03852         const bool should_not_call_base_dot_monOp = false;
03853         assert( should_not_call_base_dot_monOp );
03854 #endif
03855         (void) z;
03856         (void) x;
03857         (void) y;
03858         (void) addMonoid;
03859         (void) anyOp;
03860         (void) phase;
03861         return UNSUPPORTED;
03862     }
03863
03906     template<
03907         Descriptor descr = descriptors::no_operation,
03908         class Ring,
03909         typename IOType, typename InputType1, typename InputType2,
03910         Backend backend, typename Coords
03911     >
03912     RC dot(
03913         IOType &z,
03914         const Vector< InputType1, backend, Coords > &x,
03915         const Vector< InputType2, backend, Coords > &y,
03916         const Ring &ring = Ring(),
03917         const Phase &phase = EXECUTE,
03918         const typename std::enable_if<
03919             !grb::is_object< InputType1 >::value &&
03920             !grb::is_object< InputType2 >::value &&
03921             !grb::is_object< IOType >::value &&
03922             grb::is_semiring< Ring >::value,
03923         void >::type * const = nullptr
03924     ) {
03925 #ifdef _DEBUG
03926         std::cout << "Should not call base grb::dot (semiring version)\n";
03927 #endif
03928 #ifndef NDEBUG
03929         const bool should_not_call_base_dot_semiring = false;
03930         assert( should_not_call_base_dot_semiring );
03931 #endif
03932         (void) z;
03933         (void) x;
03934         (void) y;
03935         (void) ring;
03936         (void) phase;
03937         return UNSUPPORTED;
03938     }
03939
03942 } // end namespace grb
03943
03944 #endif // end _H_GRB_BASE_BLAS1
03945
```

10.35 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.

10.35.1 Detailed Description

Defines the ALP/GraphBLAS level-2 API.

Author

A. N. Yzelman

Date

30th of March 2017

10.36 blas2.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_BLAS2_BASE
00028 #define _H_GRB_BLAS2_BASE
00029
00030 #include <assert.h>
00031
00032 #include <graphblas/backends.hpp>
00033 #include <graphblas/blas1.hpp>
00034 #include <graphblas/descriptors.hpp>
00035 #include <graphblas/rc.hpp>
00036 #include <graphblas/semiring.hpp>
00037
00038 #include "config.hpp"
00039 #include "matrix.hpp"
00040 #include "vector.hpp"
00041
00042
00043 namespace grb {
00044
00235     template<
00236         Descriptor descr = descriptors::no_operation,
```

```
00237         class Semiring,
00238         typename IOType, typename InputType1, typename InputType2,
00239         typename InputType3, typename InputType4,
00240         typename Coords, typename RIT, typename CIT, typename NIT,
00241         Backend backend
00242     >
00243     RC mxv(
00244         Vector< IOType, backend, Coords > &u,
00245         const Vector< InputType3, backend, Coords > &u_mask,
00246         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
00247         const Vector< InputType1, backend, Coords > &v,
00248         const Vector< InputType4, backend, Coords > &v_mask,
00249         const Semiring &semiring = Semiring(),
00250         const Phase &phase = EXECUTE,
00251         const typename std::enable_if<
00252             grb::is_semiring< Semiring >::value &&
00253             !grb::is_object< IOType >::value &&
00254             !grb::is_object< InputType1 >::value &&
00255             !grb::is_object< InputType2 >::value &&
00256             !grb::is_object< InputType3 >::value &&
00257             !grb::is_object< InputType4 >::value,
00258         void >::type * const = nullptr
00259     ) {
00260 #ifdef _DEBUG
00261         std::cerr « "Selected backend does not implement mxv "
00262             « "(doubly-masked, semiring)\n";
00263 #endif
00264 #ifndef NDEBUG
00265         const bool selected_backed_does_not_support_doubly_masked_mxv_sr = false;
00266         assert( selected_backed_does_not_support_doubly_masked_mxv_sr );
00267 #endif
00268         (void) u;
00269         (void) u_mask;
00270         (void) A;
00271         (void) v;
00272         (void) v_mask;
00273         (void) semiring;
00274         return UNSUPPORTED;
00275     }
00276
00299     template<
00300         Descriptor descr = descriptors::no_operation,
00301         class Semiring,
00302         typename IOType, typename InputType1, typename InputType2,
00303         typename InputType3, typename InputType4,
00304         typename Coords, typename RIT, typename CIT, typename NIT,
00305         enum Backend backend
00306     >
00307     RC vxm(
00308         Vector< IOType, backend, Coords > &u,
00309         const Vector< InputType3, backend, Coords > &u_mask,
00310         const Vector< InputType1, backend, Coords > &v,
00311         const Vector< InputType4, backend, Coords > &v_mask,
00312         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
00313         const Semiring &semiring = Semiring(),
00314         const Phase &phase = EXECUTE,
00315         typename std::enable_if<
00316             grb::is_semiring< Semiring >::value &&
00317             !grb::is_object< InputType1 >::value &&
00318             !grb::is_object< InputType2 >::value &&
00319             !grb::is_object< InputType3 >::value &&
00320             !grb::is_object< InputType4 >::value &&
00321             !grb::is_object< IOType >::value,
00322         void >::type * = nullptr
00323     ) {
00324 #ifdef _DEBUG
00325         std::cerr « "Selected backend does not implement doubly-masked grb::vxm\n";
00326 #endif
00327 #ifndef NDEBUG
00328         const bool selected_backend_does_not_support_doubly_masked_vxm_sr = false;
00329         assert( selected_backend_does_not_support_doubly_masked_vxm_sr );
00330 #endif
00331         (void) u;
00332         (void) u_mask;
00333         (void) v;
00334         (void) v_mask;
00335         (void) A;
00336         (void) semiring;
00337         return UNSUPPORTED;
00338     }
00339
00434     template<
00435         typename Func, typename DataType,
00436         typename RIT, typename CIT, typename NIT,
00437         Backend implementation = config::default_backend,
00438         typename... Args
00439     >
```

```
00440     RC eWiseLambda(
00441         const Func f,
00442         const Matrix< DataType, implementation, RIT, CIT, NIT > &A,
00443         Args...
00444     ) {
00445 #ifdef _DEBUG
00446         std::cerr « "Selected backend does not implement grb::eWiseLambda (matrices)\n";
00447 #endif
00448 #ifndef NDEBUG
00449         const bool selected_backend_does_not_support_matrix_eWiseLamba = false;
00450         assert( selected_backend_does_not_support_matrix_eWiseLamba );
00451 #endif
00452         (void) f;
00453         (void) A;
00454         return UNSUPPORTED;
00455     }
00456 
00457      // default (non-)implementations follow:
00458 
00471     template<
00472         Descriptor descr = descriptors::no_operation,
00473         class Ring,
00474         typename IOType, typename InputType1, typename InputType2,
00475         typename InputType3,
00476         typename RIT, typename CIT, typename NIT,
00477         typename Coords,
00478         enum Backend implementation = config::default_backend
00479     >
00480     RC mxv(
00481         Vector< IOType, implementation, Coords > &u,
00482         const Vector< InputType3, implementation, Coords > &mask,
00483         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
00484         const Vector< InputType1, implementation, Coords > &v,
00485         const Ring &ring = Ring(),
00486         const Phase &phase = EXECUTE,
00487         typename std::enable_if<
00488             grb::is_semiring< Ring >::value,
00489         void >::type * = nullptr
00490     ) {
00491 #ifdef _DEBUG
00492         std::cerr « "Selected backend does not implement grb::mxv (output-masked)\n";
00493 #endif
00494 #ifndef NDEBUG
00495         const bool backend_does_not_support_output_masked_mxv = false;
00496         assert( backend_does_not_support_output_masked_mxv );
00497 #endif
00498         (void) u;
00499         (void) mask;
00500         (void) A;
00501         (void) v;
00502         (void) ring;
00503         return UNSUPPORTED;
00504     }
00505 
00518     template<
00519         Descriptor descr = descriptors::no_operation,
00520         class Ring,
00521         typename IOType, typename InputType1, typename InputType2,
00522         typename Coords, typename RIT, typename CIT, typename NIT,
00523         Backend implementation = config::default_backend
00524     >
00525     RC mxv(
00526         Vector< IOType, implementation, Coords > &u,
00527         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
00528         const Vector< InputType1, implementation, Coords > &v,
00529         const Ring &ring,
00530         typename std::enable_if<
00531             grb::is_semiring< Ring >::value, void
00532         >::type * = nullptr
00533     ) {
00534 #ifdef _DEBUG
00535         std::cerr « "Selected backend does not implement grb::mxv\n";
00536 #endif
00537 #ifndef NDEBUG
00538         const bool backend_does_not_support_mxv = false;
00539         assert( backend_does_not_support_mxv );
00540 #endif
00541         (void) u;
00542         (void) A;
00543         (void) v;
00544         (void) ring;
00545         return UNSUPPORTED;
00546     }
00547 
00560     template<
00561         Descriptor descr = descriptors::no_operation,
00562         class Ring,
```

```
00563         typename IOType, typename InputType1, typename InputType2,
00564         typename InputType3,
00565         typename Coords, typename RIT, typename CIT, typename NIT,
00566         enum Backend implementation = config::default_backend
00567     >
00568     RC vxm(
00569         Vector< IOType, implementation, Coords > &u,
00570         const Vector< InputType3, implementation, Coords > &mask,
00571         const Vector< InputType1, implementation, Coords > &v,
00572         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
00573         const Ring &ring = Ring(),
00574         const Phase &phase = EXECUTE,
00575         typename std::enable_if<
00576             grb::is_semiring< Ring >::value, void
00577         >::type * = nullptr
00578     ) {
00579 #ifdef _DEBUG
00580         std::cerr « "Selected backend does not implement grb::vxm (output-masked)\n";
00581 #endif
00582 #ifndef NDEBUG
00583         const bool selected_backend_does_not_support_output_masked_vxm = false;
00584         assert( selected_backend_does_not_support_output_masked_vxm );
00585 #endif
00586         (void) u;
00587         (void) mask;
00588         (void) v;
00589         (void) A;
00590         (void) ring;
00591         (void) phase;
00592         return UNSUPPORTED;
00593     }
00594
00607     template<
00608         Descriptor descr = descriptors::no_operation,
00609         class Ring,
00610         typename IOType, typename InputType1, typename InputType2,
00611         typename Coords, typename RIT, typename CIT, typename NIT,
00612         enum Backend implementation = config::default_backend
00613     >
00614     RC vxm(
00615         Vector< IOType, implementation, Coords > &u,
00616         const Vector< InputType1, implementation, Coords > &v,
00617         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
00618         const Ring &ring = Ring(),
00619         const Phase &phase = EXECUTE,
00620         typename std::enable_if<
00621             grb::is_semiring< Ring >::value, void
00622         >::type * = nullptr
00623     ) {
00624 #ifdef _DEBUG
00625         std::cerr « "Selected backend does not implement grb::vxm\n";
00626 #endif
00627 #ifndef NDEBUG
00628         const bool selected_backend_does_not_support_vxm = false;
00629         assert( selected_backend_does_not_support_vxm );
00630 #endif
00631         (void) u;
00632         (void) v;
00633         (void) A;
00634         (void) ring;
00635         return UNSUPPORTED;
00636     }
00637
00651     template<
00652         Descriptor descr = descriptors::no_operation,
00653         class AdditiveMonoid, class MultiplicativeOperator,
00654         typename IOType, typename InputType1, typename InputType2,
00655         typename InputType3, typename InputType4,
00656         typename Coords, typename RIT, typename CIT, typename NIT,
00657         Backend backend
00658     >
00659     RC vxm(
00660         Vector< IOType, backend, Coords > &u,
00661         const Vector< InputType3, backend, Coords > &mask,
00662         const Vector< InputType1, backend, Coords > &v,
00663         const Vector< InputType4, backend, Coords > &v_mask,
00664         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
00665         const AdditiveMonoid &add = AdditiveMonoid(),
00666         const MultiplicativeOperator &mul = MultiplicativeOperator(),
00667         const Phase &phase = EXECUTE,
00668         const typename std::enable_if<
00669             grb::is_monoid< AdditiveMonoid >::value &&
00670             grb::is_operator< MultiplicativeOperator >::value &&
00671             !grb::is_object< IOType >::value &&
00672             !grb::is_object< InputType1 >::value &&
00673             !grb::is_object< InputType2 >::value &&
00674             !grb::is_object< InputType3 >::value &&
```

```
00675             !grb::is_object< InputType4 >::value &&
00676             !std::is_same< InputType2, void >::value,
00677         void >::type * const = nullptr
00678     ) {
00679 #ifdef _DEBUG
00680         std::cerr « "Selected backend does not implement vxm (doubly-masked)\n";
00681 #endif
00682 #ifndef NDEBUG
00683         const bool selected_backed_does_not_support_doubly_masked_vxm = false;
00684         assert( selected_backed_does_not_support_doubly_masked_vxm );
00685 #endif
00686         (void) u;
00687         (void) mask;
00688         (void) v;
00689         (void) v_mask;
00690         (void) A;
00691         (void) add;
00692         (void) mul;
00693         return UNSUPPORTED;
00694     }
00695
00709     template<
00710         Descriptor descr = descriptors::no_operation,
00711         class AdditiveMonoid, class MultiplicativeOperator,
00712         typename IOType, typename InputType1, typename InputType2,
00713         typename InputType3, typename InputType4,
00714         typename Coords, typename RIT, typename CIT, typename NIT,
00715         Backend backend
00716     >
00717     RC mxv(
00718         Vector< IOType, backend, Coords > &u,
00719         const Vector< InputType3, backend, Coords > &mask,
00720         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
00721         const Vector< InputType1, backend, Coords > &v,
00722         const Vector< InputType4, backend, Coords > &v_mask,
00723         const AdditiveMonoid &add = AdditiveMonoid(),
00724         const MultiplicativeOperator &mul = MultiplicativeOperator(),
00725         const Phase &phase = EXECUTE,
00726         const typename std::enable_if<
00727             grb::is_monoid< AdditiveMonoid >::value &&
00728             grb::is_operator< MultiplicativeOperator >::value &&
00729             !grb::is_object< IOType >::value &&
00730             !grb::is_object< InputType1 >::value &&
00731             !grb::is_object< InputType2 >::value &&
00732             !grb::is_object< InputType3 >::value &&
00733             !grb::is_object< InputType4 >::value &&
00734             !std::is_same< InputType2,
00735         void >::value, void >::type * const = nullptr
00736     ) {
00737 #ifdef _DEBUG
00738         std::cerr « "Selected backend does not implement mxv (doubly-masked)\n";
00739 #endif
00740 #ifndef NDEBUG
00741         const bool selected_backed_does_not_support_doubly_masked_mxv = false;
00742         assert( selected_backed_does_not_support_doubly_masked_mxv );
00743 #endif
00744         (void) u;
00745         (void) mask;
00746         (void) A;
00747         (void) v;
00748         (void) v_mask;
00749         (void) add;
00750         (void) mul;
00751         return UNSUPPORTED;
00752     }
00753
00767     template<
00768         Descriptor descr = descriptors::no_operation,
00769         class AdditiveMonoid, class MultiplicativeOperator,
00770         typename IOType, typename InputType1, typename InputType2,
00771         typename InputType3,
00772         typename Coords, typename RIT, typename CIT, typename NIT,
00773         Backend backend
00774     >
00775     RC mxv(
00776         Vector< IOType, backend, Coords > &u,
00777         const Vector< InputType3, backend, Coords > &mask,
00778         const Matrix< InputType2, backend, RIT, NIT, CIT > &A,
00779         const Vector< InputType1, backend, Coords > &v,
00780         const AdditiveMonoid & add = AdditiveMonoid(),
00781         const MultiplicativeOperator & mul = MultiplicativeOperator(),
00782         const Phase &phase = EXECUTE,
00783         const typename std::enable_if<
00784             grb::is_monoid< AdditiveMonoid >::value &&
00785             grb::is_operator< MultiplicativeOperator >::value &&
00786             !grb::is_object< IOType >::value &&
00787             !grb::is_object< InputType1 >::value &&
```

```
00788             !grb::is_object< InputType2 >::value &&
00789             !grb::is_object< InputType3 >::value &&
00790             !std::is_same< InputType2, void >::value,
00791         void >::type * const = nullptr
00792     ) {
00793 #ifdef _DEBUG
00794         std::cerr « "Selected backend does not implement "
00795             « "singly-masked monoid-op mxv\n";
00796 #endif
00797 #ifndef NDEBUG
00798         const bool selected_backed_does_not_support_masked_monop_mxv = false;
00799         assert( selected_backed_does_not_support_masked_monop_mxv );
00800 #endif
00801         (void) u;
00802         (void) mask;
00803         (void) A;
00804         (void) v;
00805         (void) add;
00806         (void) mul;
00807         return UNSUPPORTED;
00808     }
00809
00823     template<
00824         Descriptor descr = descriptors::no_operation,
00825         class AdditiveMonoid, class MultiplicativeOperator,
00826         typename IOType, typename InputType1, typename InputType2,
00827         typename Coords, typename RIT, typename CIT, typename NIT,
00828         Backend backend
00829     >
00830     RC vxm(
00831         Vector< IOType, backend, Coords > &u,
00832         const Vector< InputType1, backend, Coords > &v,
00833         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
00834         const AdditiveMonoid &add = AdditiveMonoid(),
00835         const MultiplicativeOperator &mul = MultiplicativeOperator(),
00836         const Phase &phase = EXECUTE,
00837         const typename std::enable_if<
00838             grb::is_monoid< AdditiveMonoid >::value &&
00839             grb::is_operator< MultiplicativeOperator >::value &&
00840             !grb::is_object< IOType >::value &&
00841             !grb::is_object< InputType1 >::value &&
00842             !grb::is_object< InputType2 >::value &&
00843             !std::is_same< InputType2, void >::value,
00844         void >::type * const = nullptr
00845     ) {
00846 #ifdef _DEBUG
00847         std::cerr « "Selected backend does not implement vxm "
00848             « "(unmasked, monoid-op version )\n";
00849 #endif
00850 #ifndef NDEBUG
00851         const bool selected_backed_does_not_support_monop_vxm = false;
00852         assert( selected_backed_does_not_support_monop_vxm );
00853 #endif
00854         (void) u;
00855         (void) v;
00856         (void) A;
00857         (void) add;
00858         (void) mul;
00859         return UNSUPPORTED;
00860     }
00861
00875     template<
00876         Descriptor descr = descriptors::no_operation,
00877         class AdditiveMonoid, class MultiplicativeOperator,
00878         typename IOType, typename InputType1, typename InputType2,
00879         typename InputType3,
00880         typename Coords, typename RIT, typename CIT, typename NIT,
00881         Backend implementation
00882     >
00883     RC vxm(
00884         Vector< IOType, implementation, Coords > &u,
00885         const Vector< InputType3, implementation, Coords > &mask,
00886         const Vector< InputType1, implementation, Coords > &v,
00887         const Matrix< InputType2, implementation, RIT, CIT, NIT > &A,
00888         const AdditiveMonoid &add = AdditiveMonoid(),
00889         const MultiplicativeOperator &mul = MultiplicativeOperator(),
00890         const Phase &phase = EXECUTE,
00891         typename std::enable_if<
00892             grb::is_monoid< AdditiveMonoid >::value &&
00893             grb::is_operator< MultiplicativeOperator >::value &&
00894             !grb::is_object< IOType >::value &&
00895             !grb::is_object< InputType1 >::value &&
00896             !grb::is_object< InputType2 >::value &&
00897             !std::is_same< InputType2, void >::value,
00898         void >::type * = nullptr
00899     ) {
00900 #ifdef _DEBUG
```

```
00901         std::cerr « "Selected backend does not implement grb::vxm (output-masked)\n";
00902 #endif
00903 #ifndef NDEBUG
00904         const bool selected_backed_does_not_support_masked_monop_vxm = false;
00905         assert( selected_backed_does_not_support_masked_monop_vxm );
00906 #endif
00907         (void) u;
00908         (void) mask;
00909         (void) v;
00910         (void) A;
00911         (void) add;
00912         (void) mul;
00913         return UNSUPPORTED;
00914     }
00915
00929     template<
00930         Descriptor descr = descriptors::no_operation,
00931         class AdditiveMonoid, class MultiplicativeOperator,
00932         typename IOType, typename InputType1, typename InputType2,
00933         typename Coords, typename RIT, typename CIT, typename NIT,
00934         Backend backend
00935     >
00936     RC mxv(
00937         Vector< IOType, backend, Coords > &u,
00938         const Matrix< InputType2, backend, RIT, CIT, NIT > &A,
00939         const Vector< InputType1, backend, Coords > &v,
00940         const AdditiveMonoid &add = AdditiveMonoid(),
00941         const MultiplicativeOperator &mul = MultiplicativeOperator(),
00942         const Phase &phase = EXECUTE,
00943         const typename std::enable_if<
00944             grb::is_monoid< AdditiveMonoid >::value &&
00945             grb::is_operator< MultiplicativeOperator >::value &&
00946             !grb::is_object< IOType >::value &&
00947             !grb::is_object< InputType1 >::value &&
00948             !grb::is_object< InputType2 >::value &&
00949             !std::is_same< InputType2, void >::value,
00950         void >::type * const = nullptr
00951     ) {
00952 #ifdef _DEBUG
00953         std::cerr « "Selected backend does not implement grb::mxv (unmasked)\n";
00954 #endif
00955 #ifndef NDEBUG
00956         const bool selected_backed_does_not_support_monop_mxv = false;
00957         assert( selected_backed_does_not_support_monop_mxv );
00958 #endif
00959         (void) u;
00960         (void) A;
00961         (void) v;
00962         (void) add;
00963         (void) mul;
00964         return UNSUPPORTED;
00965     }
00966
00969 } // namespace grb
00970
00971 #endif // end _H_GRB_BLAS2_BASE
00972
```

10.37 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.*

10.37.1 Detailed Description

Defines the ALP/GraphBLAS level-3 API.

Author

A. N. Yzelman

10.38 blas3.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00026 #ifndef _H_GRB_BLAS3_BASE
00027 #define _H_GRB_BLAS3_BASE
00028
00029 #include <graphblas/backends.hpp>
00030 #include <graphblas/phase.hpp>
00031
00032 #include "matrix.hpp"
00033 #include "vector.hpp"
00034
00035
00036 namespace grb {
00037
00087     template<
00088         Descriptor descr = descriptors::no_operation,
00089         typename OutputType, typename InputType1, typename InputType2,
00090         typename CIT, typename RIT, typename NIT,
00091         class Semiring,
00092         Backend backend
00093     >
00094     RC mxm(
00095         Matrix< OutputType, backend, CIT, RIT, NIT > &C,
00096         const Matrix< InputType1, backend, CIT, RIT, NIT > &A,
00097         const Matrix< InputType2, backend, CIT, RIT, NIT > &B,
00098         const Semiring &ring = Semiring(),
00099         const Phase &phase = EXECUTE
00100     ) {
00101 #ifdef _DEBUG
```

```
00102         std::cerr « "Selected backend does not implement grb::mxm "
00103             « "(semiring version)\n";
00104 #endif
00105 #ifndef NDEBUG
00106         const bool selected_backend_does_not_support_mxm = false;
00107         assert( selected_backend_does_not_support_mxm );
00108 #endif
00109         (void) C;
00110         (void) A;
00111         (void) B;
00112         (void) ring;
00113         (void) phase;
00114         // this is the generic stub implementation
00115         return UNSUPPORTED;
00116     }
00117
00175     template<
00176         Descriptor descr = descriptors::no_operation,
00177         typename OutputType, typename InputType1, typename InputType2,
00178         typename InputType3, typename RIT, typename CIT, typename NIT,
00179         Backend backend, typename Coords
00180     >
00181     RC zip(
00182         Matrix< OutputType, backend, RIT, CIT, NIT > &A,
00183         const Vector< InputType1, backend, Coords > &x,
00184         const Vector< InputType2, backend, Coords > &y,
00185         const Vector< InputType3, backend, Coords > &z,
00186         const Phase &phase = EXECUTE
00187     ) {
00188         (void) x;
00189         (void) y;
00190         (void) z;
00191         (void) phase;
00192 #ifdef _DEBUG
00193         std::cerr « "Selected backend does not implement grb::zip (vectors into "
00194             « "matrices, non-void)\n";
00195 #endif
00196 #ifndef NDEBUG
00197         const bool selected_backend_does_not_support_zip_from_vectors_to_matrix
00198             = false;
00199         assert( selected_backend_does_not_support_zip_from_vectors_to_matrix );
00200 #endif
00201         const RC ret = grb::clear( A );
00202         return ret == SUCCESS ? UNSUPPORTED : ret;
00203     }
00204
00217     template<
00218         Descriptor descr = descriptors::no_operation,
00219         typename InputType1, typename InputType2, typename InputType3,
00220         typename RIT, typename CIT, typename NIT,
00221         Backend backend, typename Coords
00222     >
00223     RC zip(
00224         Matrix< void, backend, RIT, CIT, NIT > &A,
00225         const Vector< InputType1, backend, Coords > &x,
00226         const Vector< InputType2, backend, Coords > &y,
00227         const Phase &phase = EXECUTE
00228     ) {
00229         (void) x;
00230         (void) y;
00231         (void) phase;
00232 #ifdef _DEBUG
00233         std::cerr « "Selected backend does not implement grb::zip (vectors into "
00234             « "matrices, void)\n";
00235 #endif
00236 #ifndef NDEBUG
00237         const bool selected_backend_does_not_support_zip_from_vectors_to_void_matrix
00238             = false;
00239         assert( selected_backend_does_not_support_zip_from_vectors_to_void_matrix );
00240 #endif
00241         const RC ret = grb::clear( A );
00242         return ret == SUCCESS ? UNSUPPORTED : ret;
00243     }
00244
00249 } // namespace grb
00250
00251 #endif // end _H_GRB_BLAS3_BASE
00252
```

10.39 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.

10.39.1 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

10.40 collectives.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00028 #ifndef _H_GRB_COLL_BASE
00029 #define _H_GRB_COLL_BASE
00030
00031 #include <graphblas/backends.hpp>
00032 #include <graphblas/descriptors.hpp>
00033 #include <graphblas/rc.hpp>
00034
00035
00036 namespace grb {
00037
00045     template< enum Backend implementation >
00046     class collectives {
00047
00048         private:
00049
00051             collectives() {}
00052
00053
00054         public:
00055
00116             template<
00117                 Descriptor descr = descriptors::no_operation,
00118                 typename Operator,
```

```
00119                 typename IOType
00120             >
00121             static RC allreduce( IOType &inout, const Operator op = Operator() ) {
00122                 (void) inout;
00123                 (void)op;
00124                 return PANIC;
00125             }
00126
00186             template<
00187                 Descriptor descr = descriptors::no_operation,
00188                 typename Operator,
00189                 typename IOType
00190             >
00191             static RC reduce(
00192                 IOType &inout,
00193                 const size_t root = 0,
00194                 const Operator op = Operator()
00195             ) {
00196                 (void) inout;
00197                 (void) op;
00198                 (void) root;
00199                 return PANIC;
00200             }
00201
00243             template< typename IOType >
00244             static RC broadcast( IOType &inout, const size_t root = 0 ) {
00245                 (void) inout;
00246                 (void) root;
00247                 return PANIC;
00248             }
00249
00256             template< Descriptor descr = descriptors::no_operation, typename IOType >
00257             static RC broadcast(
00258                 IOType * inout,
00259                 const size_t size,
00260                 const size_t root = 0
00261             ) {
00262                 (void) inout;
00263                 (void) size;
00264                 (void) root;
00265                 return PANIC;
00266             }
00267
00268     }; // end class "collectives"
00269
00270 } // end namespace grb
00271
00272 #endif // end _H_GRB_COLL_BASE
00273
```

10.41 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 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.

10.41.1 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

10.42 base/config.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00028 #ifndef _H_GRB_CONFIG_BASE
00029 #define _H_GRB_CONFIG_BASE
00030
00031 #include <cstddef> //size_t
00032 #include <string>
00033
00034 #include <assert.h>
```

```
00035 #include <unistd.h> //sysconf
00036
00037 #include <graphblas/backends.hpp>
00038
00039 #ifndef _GRB_NO_STDIO
00040  #include <iostream> //std::cout
00041 #endif
00042
00043 // if the user did not define _GRB_BACKEND, set it to the default sequential
00044 // implementation
00045 #ifndef _GRB_BACKEND
00046  #define _GRB_BACKEND reference
00047 #endif
00048
00049
00050 namespace grb {
00051
00056     namespace config {
00057
00079         static constexpr grb::Backend default_backend = _GRB_BACKEND;
00080
00086         class CACHE_LINE_SIZE {
00087
00088             private:
00089
00094                 static constexpr size_t bytes = 64;
00095
00096             public:
00097
00104                 static constexpr size_t value() {
00105                     return bytes;
00106                 }
00107
00108         };
00109
00115         class SIMD_SIZE {
00116
00117             private:
00118
00123                 static constexpr size_t bytes = 64;
00124
00125             public:
00126
00133                 static constexpr size_t value() {
00134                     return bytes;
00135                 }
00136
00137         };
00138
00145         template< typename T >
00146         class SIMD_BLOCKSIZE {
00147
00148             public:
00149
00158                 static constexpr size_t unsafe_value() {
00159                     return SIMD_SIZE::value() / sizeof( T );
00160                 }
00161
00168                 static constexpr size_t value() {
00169                     return unsafe_value() > 0 ? unsafe_value() : 1;
00170                 }
00171
00172         };
00173
00185         class HARDWARE_THREADS {
00186
00187             public:
00188
00200                 static long value() {
00201                     return sysconf( _SC_NPROCESSORS_ONLN );
00202                 }
00203
00204         };
00205
00211         class BENCHMARKING {
00212
00213             public:
00214
00216                 static constexpr size_t inner() {
00217                     return 1;
00218                 }
00219
00221                 static constexpr size_t outer() {
00222                     return 10;
00223                 }
00224
00225         };
```

```
00226
00232         class MEMORY {
00233
00234             public:
00235
00237                 static constexpr size_t l1_cache_size() {
00238                     return 32768;
00239                 }
00240
00244                 static constexpr size_t big_memory() {
00245                     return 31;
00246                 } // 2GB
00247
00269                 static constexpr double random_access_memspeed() {
00270                     return 147.298;
00271                 }
00272
00294                 static constexpr double stream_memspeed() {
00295                     return 1931.264;
00296                 }
00297
00305                 static bool report(
00306                     const std::string prefix, const std::string action,
00307                     const size_t size, const bool printNewline = true
00308                 ) {
00309 #ifdef _GRB_NO_STDIO
00310                     (void) prefix;
00311                     (void) action;
00312                     (void) size;
00313                     (void) printNewline;
00314                     return false;
00315 #else
00316                     constexpr size_t big =
00317  #ifdef _DEBUG
00318                         true;
00319  #else
00320                         ( 1ul << big_memory() );
00321  #endif
00322                     if( size >= big ) {
00323                         std::cout << "Info: ";
00324                         std::cout << prefix << " ";
00325                         std::cout << action << " ";
00326                         if( sizeof( size_t ) * 8 > 40 && ( size >> 40 ) > 2 ) {
00327                             std::cout << ( size >> 40 ) << " TB of memory";
00328                         } else if( sizeof( size_t ) * 8 > 30 && ( size >> 30 ) > 2 ) {
00329                             std::cout << ( size >> 30 ) << " GB of memory";
00330                         } else if( sizeof( size_t ) * 8 > 20 && ( size >> 20 ) > 2 ) {
00331                             std::cout << ( size >> 20 ) << " MB of memory";
00332                         } else if( sizeof( size_t ) * 8 > 10 && ( size >> 10 ) > 2 ) {
00333                             std::cout << ( size >> 10 ) << " kB of memory";
00334                         } else {
00335                             std::cout << size << " bytes of memory";
00336                         }
00337                         if( printNewline ) {
00338                             std::cout << ".\n";
00339                         }
00340                         return true;
00341                     }
00342                     return false;
00343 #endif
00344                 }
00345         };
00346
00357         template< grb::Backend implementation = default_backend >
00358         class IMPLEMENTATION {};
00359
00371         typedef unsigned int RowIndexType;
00372
00384         typedef unsigned int ColIndexType;
00385
00397         typedef size_t NonzeroIndexType;
00398
00410         typedef unsigned int VectorIndexType;
00411
00412     } // namespace config
00413
00414 } // namespace grb
00415
00416 #endif // end _H_GRB_CONFIG_BASE
00417
```

10.43 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.

10.43.1 Detailed Description

Contains the configuration parameters for the BSP1D backend.

Author

A. N. Yzelman

Date

5th of May, 2017

10.44 bsp1d/config.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_BSP1D_CONFIG
00028 #define _H_GRB_BSP1D_CONFIG
00029
00030 #include <assert.h>
00031
00032 #include <graphblas/reference/config.hpp>
00033
00034 // if not defined, we set the backend of the BSP1D implementation to the
00035 // reference implementation
00036 #ifndef _GRB_BSP1D_BACKEND
```

```
00037  #pragma message "_GRB_BSP1D_BACKEND was not set-- auto-selecting reference"
00038  #define _GRB_BSP1D_BACKEND reference
00039 #endif
00040
00041
00042 namespace grb {
00043
00044     namespace config {
00045
00061         template<>
00062         class IMPLEMENTATION< BSP1D > {
00063
00064             private:
00065
00071                 static bool set;
00072
00078                 static grb::config::ALLOC_MODE mode;
00079
00085                 static void deduce() noexcept;
00086
00087
00088             public:
00089
00093                 static constexpr ALLOC_MODE defaultAllocMode() {
00094                     return grb::config::ALLOC_MODE::ALIGNED;
00095                 }
00096
00103                 static constexpr bool fixedVectorCapacities() {
00104                     return IMPLEMENTATION< _GRB_BSP1D_BACKEND >::fixedVectorCapacities();
00105                 }
00106
00124                 static grb::config::ALLOC_MODE sharedAllocMode() noexcept;
00125
00131                 static constexpr Backend coordinatesBackend() {
00132                     return IMPLEMENTATION< _GRB_BSP1D_BACKEND >::coordinatesBackend();
00133                 }
00134
00135         };
00136
00139     } // namespace config
00140
00141 } // namespace grb
00142
00143 #endif // end "_H_GRB_BSP1D_CONFIG"
00144
```

10.45 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.

10.45.1 Detailed Description

Contains the configuration parameters for the reference and reference_omp backends.

Author

A. N. Yzelman

Date

13th of September 2017.

10.46 reference/config.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00028 #ifndef _H_GRB_REFERENCE_CONFIG
00029 #define _H_GRB_REFERENCE_CONFIG
00030
00031 #include <graphblas/base/config.hpp>
00032
00033
00034 namespace grb {
00035
00036     namespace config {
00037
00054         enum ALLOC_MODE {
00055
00057             ALIGNED,
00058
00060             INTERLEAVED
00061
00062         };
00063
00070         std::string toString( const ALLOC_MODE mode );
00071
00096         template< Backend backend >
00097         class PREFETCHING {
00098
```

```
00099             // guard against unintended use
00100             static_assert( backend == reference || backend == reference_omp,
00101                 "Instantiating for non-reference backend" );
00102
00103             public:
00104
00108                 static constexpr bool enabled() {
00109                     return false;
00110                 }
00111
00117                 static constexpr size_t distance() {
00118                     return 128;
00119                 }
00120
00121         };
00122
00131         template<>
00132         class IMPLEMENTATION< reference > {
00133
00134             public:
00135
00137                 static constexpr ALLOC_MODE defaultAllocMode() {
00138                     return ALLOC_MODE::ALIGNED;
00139                 }
00140
00142                 static constexpr ALLOC_MODE sharedAllocMode() {
00143                     return ALLOC_MODE::ALIGNED;
00144                 }
00145
00152                 static constexpr bool fixedVectorCapacities() {
00153                     return true;
00154                 }
00155
00164                 static constexpr size_t vectorBufferSize( const size_t, const size_t ) {
00165                     return 0;
00166                 }
00167
00173                 static constexpr Backend coordinatesBackend() {
00174                     return reference;
00175                 }
00176
00177         };
00178
00187         template<>
00188         class IMPLEMENTATION< reference_omp > {
00189
00190             private:
00191
00198                 static constexpr size_t minVectorBufferChunksize() {
00199                     return CACHE_LINE_SIZE::value();
00200                 }
00201
00216                 static constexpr size_t absVectorBufferSize() {
00217                     return 0;
00218                 }
00219
00236                 static constexpr double relVectorBufferSize() {
00237                     return 1;
00238                 }
00239
00240
00241         public:
00242
00247                 static constexpr ALLOC_MODE defaultAllocMode() {
00248                     return ALLOC_MODE::ALIGNED;
00249                 }
00250
00255                 static constexpr ALLOC_MODE sharedAllocMode() {
00256                     // return ALLOC_MODE::ALIGNED; //DBG
00257                     return ALLOC_MODE::INTERLEAVED;
00258                 }
00259
00265                 static constexpr Backend coordinatesBackend() {
00266                     return reference_omp;
00267                 }
00268
00275                 static constexpr bool fixedVectorCapacities() {
00276                     return true;
00277                 }
00278
00295                 static inline size_t vectorBufferSize( const size_t n, const size_t T ) {
00296                     size_t ret;
00297                     if( absVectorBufferSize() > 0 ) {
00298                         (void)n;
00299                         ret = absVectorBufferSize();
00300                     } else {
00301                         constexpr const double factor = relVectorBufferSize();
```

```
00302                         static_assert( factor > 0, "Configuration error" );
00303                         ret = factor * n;
00304                     }
00305                     ret = std::max( ret, T * minVectorBufferChunksize() );
00306                     ret += T;
00307                     if( ret % T > 0 ) {
00308                         ret += T - ( ret % T );
00309                     }
00310                     ret = std::max( 2 * T, ret );
00311                     assert( ret % T == 0 );
00312                     return ret;
00313                 }
00314
00315         };
00316
00319     } // namespace config
00320
00321 } // namespace grb
00322
00323 #endif // end "_H_GRB_REFERENCE_CONFIG"
00324
```

10.47 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.

10.47.1 Detailed Description

Specifies the grb::Launcher functionalities.

Author

A. N. Yzelman

Date

17th of April, 2017

10.48 exec.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_EXEC_BASE
00028 #define _H_GRB_EXEC_BASE
00029
00030 #include <stdexcept>
00031 #include <string>
00032
00033 #include <graphblas/backends.hpp>
00034 #include <graphblas/rc.hpp>
00035
00036 #ifndef _GRB_NO_STDIO
00037  #include <iostream>
00038 #endif
00039
00040
00041 namespace grb {
00042
00056     enum EXEC_MODE {
00057
00062         AUTOMATIC = 0,
00063
00069         MANUAL,
00070
00076         FROM_MPI
00077
00078     };
00079
00096     template< enum EXEC_MODE mode, enum Backend backend >
00097     class Launcher {
00098
00099         public :
00100
00140             Launcher(
00141                 const size_t process_id = 0,
00142                 const size_t nprocs = 1,
00143                 const std::string hostname = "localhost",
00144                 const std::string port = "0"
00145             ) {
00146                 // spec does not specify any constrants on hostname and port
00147                 // so accept (and ignore) anything
00148                 (void) hostname; (void) port;
00149
00150 #ifndef _GRB_NO_EXCEPTIONS
00151                 // sanity checks on process_id and nprocs
00152                 if( nprocs == 0 ) {
00153                     throw std::invalid_argument( "Total number of user processes must be "
00154                         "strictly larger than zero." );
00155                 }
00156                 if( process_id >= nprocs ) {
00157                     throw std::invalid_argument( "Process ID must be strictly smaller than "
00158                         "total number of user processes." );
00159                 }
00160 #endif
00161             }
00162
00209             template< typename T, typename U >
00210             RC exec(
00211                 void ( *alp_program )( const T &, U & ),
00212                 const T &data_in,
00213                 U &data_out,
00214                 const bool broadcast = false
00215             ) const {
00216                 (void) alp_program;
00217                 (void) data_in;
00218                 (void) data_out;
00219                 (void) broadcast;
```

```
00220                 // stub implementation, should be overridden by specialised backend,
00221                 // so return error code
00222                 return PANIC;
00223             }
00224
00251             template< typename U >
00252             RC exec(
00253                 void ( *alp_program )( const void *, const size_t, U & ),
00254                 const void * data_in,
00255                 const size_t in_size,
00256                 U &data_out,
00257                 const bool broadcast = false
00258             ) const {
00259                 (void) alp_program;
00260                 (void) data_in;
00261                 (void) in_size;
00262                 (void) data_out;
00263                 (void) broadcast;
00264                 return PANIC;
00265             }
00266
00322             static RC finalize() {
00323                 return PANIC;
00324             }
00325
00326     }; // end class 'Launcher'
00327
00328 } // end namespace "grb"
00329
00330 #endif // end _H_GRB_EXEC_BASE
00331
```

10.49 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.

10.49.1 Detailed Description

Specifies the grb::init and grb::finalize functionalities.

Author

A. N. Yzelman

Date

24th of January, 2017

10.50 init.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_INIT_BASE
00028 #define _H_GRB_INIT_BASE
00029
00030 #include <graphblas/rc.hpp>
00031
00032 #include "config.hpp"
00033
00034
00035 namespace grb {
00036
00144     template< enum Backend backend = config::default_backend >
00145     RC init( const size_t s, const size_t P, void * const implementation_data ) {
00146         (void) s;
00147         (void) P;
00148         (void) implementation_data;
00149         return PANIC;
00150     }
00151
00171     template< enum Backend backend = config::default_backend >
00172     RC init() {
00173         return grb::init< backend >( 0, 1, nullptr );
00174     }
00175
00209     template< enum Backend backend = config::default_backend >
00210     RC finalize() {
00211         return PANIC;
00212     }
00213
00214 } // namespace grb
00215
00216 #endif // end _H_GRB_INIT_BASE
00217
```

10.51 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.

10.51.1 Detailed Description

Specifies all I/O primitives for use with ALP/GraphBLAS containers.

Author

A. N. Yzelman

Date

21st of February, 2017

10.52 io.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #include <type_traits>
00028 #include <typeinfo>
00029
00030 #ifndef _H_GRB_IO_BASE
00031 #define _H_GRB_IO_BASE
00032
00033 #include <graphblas/rc.hpp>
00034 #include <graphblas/phase.hpp>
00035 #include <graphblas/iomode.hpp>
00036 #include <graphblas/SynchronizedNonzeroIterator.hpp>
00037 #include <graphblas/utils/iterators/type_traits.hpp>
00038
00039 #include "matrix.hpp"
00040 #include "vector.hpp"
00041
00042 #include <assert.h>
00043
00044
00045 namespace grb {
00046
00153     template<
00154         typename ElementType, typename Coords,
00155         Backend implementation = config::default_backend
00156     >
00157     uintptr_t getID(
00158         const Vector< ElementType, implementation, Coords > &x
00159     ) {
00160         (void) x;
00161 #ifndef NDEBUG
00162         const bool this_is_an_invalid_default_implementation = false;
00163 #endif
00164         assert( this_is_an_invalid_default_implementation );
00165         return static_cast< uintptr_t >(-1);
```

```
00166     }
00167
00174     template<
00175         typename ElementType, typename RIT, typename CIT, typename NIT,
00176         Backend implementation = config::default_backend
00177     >
00178     uintptr_t getID(
00179         const Matrix< ElementType, implementation, RIT, CIT, NIT > &x
00180     ) {
00181         (void) x;
00182 #ifndef NDEBUG
00183         const bool this_is_an_invalid_default_implementation = false;
00184 #endif
00185         assert( this_is_an_invalid_default_implementation );
00186         return static_cast< uintptr_t >(-1);
00187     }
00188
00231     template<
00232         typename DataType,
00233         Backend backend, typename Coords
00234     >
00235     size_t size( const Vector< DataType, backend, Coords > &x ) noexcept {
00236 #ifndef NDEBUG
00237         const bool may_not_call_base_size = false;
00238 #endif
00239         (void) x;
00240         assert( may_not_call_base_size );
00241         return SIZE_MAX;
00242     }
00243
00282     template<
00283         typename InputType, Backend backend,
00284         typename RIT, typename CIT, typename NIT
00285     >
00286     size_t nrows(
00287         const Matrix< InputType, backend, RIT, CIT, NIT > &A
00288     ) noexcept {
00289 #ifndef NDEBUG
00290         const bool may_not_call_base_nrows = false;
00291 #endif
00292         (void) A;
00293         assert( may_not_call_base_nrows );
00294         return SIZE_MAX;
00295     }
00296
00335     template<
00336         typename InputType, Backend backend,
00337         typename RIT, typename CIT, typename NIT
00338     >
00339     size_t ncols(
00340         const Matrix< InputType, backend, RIT, CIT, NIT > &A
00341     ) noexcept {
00342 #ifndef NDEBUG
00343         const bool may_not_call_base_ncols = false;
00344 #endif
00345         (void) A;
00346         assert( may_not_call_base_ncols );
00347         return SIZE_MAX;
00348     }
00349
00385     template<
00386         typename InputType, Backend backend, typename Coords
00387     >
00388     size_t capacity( const Vector< InputType, backend, Coords > &x ) noexcept {
00389 #ifndef NDEBUG
00390         const bool should_not_call_base_vector_capacity = false;
00391 #endif
00392         assert( should_not_call_base_vector_capacity );
00393         (void) x;
00394         return SIZE_MAX;
00395     }
00396
00428     template<
00429         typename InputType, Backend backend,
00430         typename RIT, typename CIT, typename NIT
00431     >
00432     size_t capacity(
00433         const Matrix< InputType, backend, RIT, CIT, NIT > &A
00434     ) noexcept {
00435 #ifndef NDEBUG
00436         const bool should_not_call_base_matrix_capacity = false;
00437 #endif
00438         assert( should_not_call_base_matrix_capacity );
00439         (void) A;
00440         return SIZE_MAX;
00441     }
00442
```

```
00478     template< typename DataType, Backend backend, typename Coords >
00479     size_t nnz( const Vector< DataType, backend, Coords > &x ) noexcept {
00480 #ifndef NDEBUG
00481         const bool should_not_call_base_vector_nnz = false;
00482 #endif
00483         (void) x;
00484         assert( should_not_call_base_vector_nnz );
00485         return SIZE_MAX;
00486     }
00487
00519     template<
00520         typename InputType, Backend backend,
00521         typename RIT, typename CIT, typename NIT
00522     >
00523     size_t nnz(
00524         const Matrix< InputType, backend, RIT, CIT, NIT > &A
00525     ) noexcept {
00526 #ifndef NDEBUG
00527         const bool should_not_call_base_matrix_nnz = false;
00528 #endif
00529         (void) A;
00530         assert( should_not_call_base_matrix_nnz );
00531         return SIZE_MAX;
00532     }
00533
00573     template< typename DataType, Backend backend, typename Coords >
00574     RC clear( Vector< DataType, backend, Coords > &x ) noexcept {
00575 #ifndef NDEBUG
00576         const bool should_not_call_base_vector_clear = false;
00577 #endif
00578         (void) x;
00579         assert( should_not_call_base_vector_clear );
00580         return UNSUPPORTED;
00581     }
00582
00618     template<
00619         typename InputType, Backend backend,
00620         typename RIT, typename CIT, typename NIT
00621     >
00622     RC clear(
00623         Matrix< InputType, backend, RIT, CIT, NIT > &A
00624     ) noexcept {
00625 #ifndef NDEBUG
00626         const bool should_not_call_base_matrix_clear = false;
00627 #endif
00628         (void) A;
00629         assert( should_not_call_base_matrix_clear );
00630         return UNSUPPORTED;
00631     }
00632
00699     template<
00700         typename InputType,
00701         Backend backend, typename Coords
00702     >
00703     RC resize(
00704         Vector< InputType, backend, Coords > &x,
00705         const size_t new_nz
00706     ) noexcept {
00707 #ifndef NDEBUG
00708         const bool should_not_call_base_vector_resize = false;
00709 #endif
00710         (void) x;
00711         (void) new_nz;
00712         assert( should_not_call_base_vector_resize );
00713         return UNSUPPORTED;
00714     }
00715
00782     template<
00783         typename InputType, Backend backend,
00784         typename RIT, typename CIT, typename NIT
00785     >
00786     RC resize(
00787         Matrix< InputType, backend, RIT, CIT, NIT > &A, const size_t new_nz
00788     ) noexcept {
00789 #ifndef NDEBUG
00790         const bool should_not_call_base_matrix_resize = false;
00791 #endif
00792         (void) A;
00793         (void) new_nz;
00794         assert( should_not_call_base_matrix_resize );
00795         return UNSUPPORTED;
00796     }
00797
00852     template<
00853         Descriptor descr = descriptors::no_operation,
00854         typename DataType, typename T,
00855         typename Coords, Backend backend
```

```
00856     >
00857     RC set(
00858         Vector< DataType, backend, Coords > &x, const T val,
00859         const Phase &phase = EXECUTE,
00860         const typename std::enable_if<
00861             !grb::is_object< DataType >::value &&
00862             !grb::is_object< T >::value,
00863         void >::type * const = nullptr
00864     ) noexcept {
00865 #ifndef NDEBUG
00866         const bool should_not_call_base_vector_set = false;
00867         assert( should_not_call_base_vector_set );
00868 #endif
00869         (void) x;
00870         (void) val;
00871         (void) phase;
00872         return UNSUPPORTED;
00873     }
00874
00936     template<
00937         Descriptor descr = descriptors::no_operation,
00938         typename DataType, typename MaskType, typename T,
00939         Backend backend, typename Coords
00940     >
00941     RC set(
00942         Vector< DataType, reference, Coords > &x,
00943         const Vector< MaskType, backend, Coords > &mask,
00944         const T val,
00945         const Phase &phase = EXECUTE,
00946         const typename std::enable_if<
00947             !grb::is_object< DataType >::value && !grb::is_object< T >::value,
00948         void >::type * const = nullptr
00949     ) {
00950 #ifndef NDEBUG
00951         const bool should_not_call_base_masked_vector_set = false;
00952         assert( should_not_call_base_masked_vector_set );
00953 #endif
00954         (void) x;
00955         (void) mask;
00956         (void) val;
00957         (void) phase;
00958         return UNSUPPORTED;
00959     }
00960
00993     template<
00994         Descriptor descr = descriptors::no_operation,
00995         typename OutputType, typename InputType,
00996         Backend backend, typename Coords
00997     >
00998     RC set(
00999         Vector< OutputType, backend, Coords > &x,
01000         const Vector< InputType, backend, Coords > &y,
01001         const Phase &phase = EXECUTE
01002     ) {
01003 #ifndef NDEBUG
01004         const bool should_not_call_base_vector_set_copy = false;
01005         assert( should_not_call_base_vector_set_copy );
01006 #endif
01007         (void) x;
01008         (void) y;
01009         (void) phase;
01010         return UNSUPPORTED;
01011     }
01012
01054     template<
01055         Descriptor descr = descriptors::no_operation,
01056         typename OutputType, typename MaskType, typename InputType,
01057         Backend backend, typename Coords
01058     >
01059     RC set(
01060         Vector< OutputType, backend, Coords > &x,
01061         const Vector< MaskType, backend, Coords > &mask,
01062         const Vector< InputType, backend, Coords > &y,
01063         const Phase &phase = EXECUTE,
01064         const typename std::enable_if< !grb::is_object< OutputType >::value &&
01065             !grb::is_object< MaskType >::value &&
01066             !grb::is_object< InputType >::value,
01067         void >::type * const = nullptr
01068     ) {
01069 #ifndef NDEBUG
01070         const bool should_not_call_base_vector_set_copy_masked = false;
01071         assert( should_not_call_base_vector_set_copy_masked );
01072 #endif
01073         (void) x;
01074         (void) mask;
01075         (void) y;
01076         (void) phase;
```

```
01077         return UNSUPPORTED;
01078     }
01079
01123     template<
01124         Descriptor descr = descriptors::no_operation,
01125         typename DataType, typename T,
01126         Backend backend, typename Coords
01127     >
01128     RC setElement( Vector< DataType, backend, Coords > &x,
01129         const T val,
01130         const size_t i,
01131         const Phase &phase = EXECUTE,
01132         const typename std::enable_if< !grb::is_object< DataType >::value &&
01133             !grb::is_object< T >::value, void >::type * const = nullptr
01134     ) {
01135 #ifndef NDEBUG
01136         const bool should_not_call_base_setElement = false;
01137         assert( should_not_call_base_setElement );
01138 #endif
01139         (void) x;
01140         (void) val;
01141         (void) i;
01142         (void) phase;
01143         return UNSUPPORTED;
01144     }
01145
01152     template<
01153         Descriptor descr = descriptors::no_operation,
01154         typename InputType, typename fwd_iterator,
01155         Backend backend, typename Coords
01156     >
01157     RC buildVector(
01158         Vector< InputType, backend, Coords > &x,
01159         fwd_iterator start, const fwd_iterator end,
01160         const IOMode mode
01161     ) {
01162         operators::right_assign< InputType > accum;
01163         return buildVector< descr >( x, accum, start, end, mode );
01164     }
01165
01173     template< Descriptor descr = descriptors::no_operation,
01174         typename InputType,
01175         class Merger = operators::right_assign< InputType >,
01176         typename fwd_iterator1, typename fwd_iterator2,
01177         Backend backend, typename Coords
01178     >
01179     RC buildVector( Vector< InputType, backend, Coords > &x,
01180         fwd_iterator1 ind_start, const fwd_iterator1 ind_end,
01181         fwd_iterator2 val_start, const fwd_iterator2 val_end,
01182         const IOMode mode, const Merger & merger = Merger()
01183     ) {
01184         operators::right_assign< InputType > accum;
01185         return buildVector< descr >( x, accum, ind_start, ind_end, val_start, val_end,
01186             mode, merger );
01187     }
01188
01220     template<
01221         Descriptor descr = descriptors::no_operation,
01222         typename InputType,
01223         class Merger = operators::right_assign< InputType >,
01224         typename fwd_iterator1, typename fwd_iterator2,
01225         Backend backend, typename Coords
01226     >
01227     RC buildVectorUnique(
01228         Vector< InputType, backend, Coords > &x,
01229         fwd_iterator1 ind_start, const fwd_iterator1 ind_end,
01230         fwd_iterator2 val_start, const fwd_iterator2 val_end,
01231         const IOMode mode
01232     ) {
01233         return buildVector< descr | descriptors::no_duplicates >(
01234             x,
01235             ind_start, ind_end,
01236             val_start, val_end,
01237             mode
01238         );
01239     }
01240
01327     template<
01328         Descriptor descr = descriptors::no_operation,
01329         typename InputType,
01330         typename fwd_iterator1 = const size_t * __restrict__,
01331         typename fwd_iterator2 = const size_t * __restrict__,
01332         typename fwd_iterator3 = const InputType * __restrict__,
01333         Backend implementation = config::default_backend
01334     >
01335     RC buildMatrixUnique(
01336         Matrix< InputType, implementation > &A,
```

```
01337         fwd_iterator1 I, const fwd_iterator1 I_end,
01338         fwd_iterator2 J, const fwd_iterator2 J_end,
01339         fwd_iterator3 V, const fwd_iterator3 V_end,
01340         const IOMode mode
01341     ) {
01342         // derive synchronized iterator
01343         auto start = internal::makeSynchronized(
01344             I, J, V,
01345             I_end, J_end, V_end
01346         );
01347         const auto end = internal::makeSynchronized(
01348             I_end, J_end, V_end,
01349             I_end, J_end, V_end
01350         );
01351         // defer to other signature
01352         return buildMatrixUnique< descr >( A, start, end, mode );
01353     }
01354
01359     template<
01360         Descriptor descr = descriptors::no_operation,
01361         typename InputType,
01362         typename fwd_iterator1 = const size_t * __restrict__,
01363         typename fwd_iterator2 = const size_t * __restrict__,
01364         typename fwd_iterator3 = const InputType * __restrict__,
01365         Backend implementation = config::default_backend
01366     >
01367     RC buildMatrixUnique(
01368         Matrix< InputType, implementation > &A,
01369         fwd_iterator1 I, fwd_iterator2 J, fwd_iterator3 V,
01370         const size_t nz, const IOMode mode
01371     ) {
01372         return buildMatrixUnique< descr >(
01373             A,
01374             I, I + nz,
01375             J, J + nz,
01376             V, V + nz,
01377             mode
01378         );
01379     }
01380
01385     template<
01386         Descriptor descr = descriptors::no_operation,
01387         typename InputType, typename RIT, typename CIT, typename NIT,
01388         typename fwd_iterator1 = const size_t * __restrict__,
01389         typename fwd_iterator2 = const size_t * __restrict__,
01390         typename length_type = size_t,
01391         Backend implementation = config::default_backend
01392     >
01393     RC buildMatrixUnique(
01394         Matrix< InputType, implementation, RIT, CIT, NIT > &A,
01395         fwd_iterator1 I, fwd_iterator2 J,
01396         const length_type nz, const IOMode mode
01397     ) {
01398         // derive synchronized iterator
01399         auto start = internal::makeSynchronized( I, J, I + nz, J + nz );
01400         const auto end = internal::makeSynchronized(
01401             I + nz, J + nz, I + nz, J + nz );
01402         // defer to other signature
01403         return buildMatrixUnique< descr >( A, start, end, mode );
01404     }
01405
01447     template<
01448         Descriptor descr = descriptors::no_operation,
01449         typename InputType, typename RIT, typename CIT, typename NIT,
01450         typename fwd_iterator,
01451         Backend implementation = config::default_backend
01452     >
01453     RC buildMatrixUnique(
01454         Matrix< InputType, implementation, RIT, CIT, NIT > &A,
01455         fwd_iterator start, const fwd_iterator end,
01456         const IOMode mode
01457     ) {
01458         (void) A;
01459         (void) start;
01460         (void) end;
01461         (void) mode;
01462 #ifndef NDEBUG
01463         std::cerr « "Should not call base grb::buildMatrixUnique" « std::endl;
01464         const bool should_not_call_base_buildMatrixUnique = false;
01465         assert( should_not_call_base_buildMatrixUnique );
01466 #endif
01467         return PANIC;
01468     }
01469
01516     template< Backend backend = config::default_backend >
01517     RC wait() {
01518 #ifndef NDEBUG
```

```
01519         const bool should_not_call_base_wait = false;
01520         assert( should_not_call_base_wait );
01521 #endif
01522         return UNSUPPORTED;
01523     }
01524
01550     template<
01551         Backend backend, typename InputType, typename Coords,
01552         typename... Args
01553     >
01554     RC wait(
01555         const Vector< InputType, backend, Coords > &x,
01556         const Args &... args
01557     ) {
01558 #ifndef NDEBUG
01559         const bool should_not_call_base_vector_wait = false;
01560         assert( should_not_call_base_vector_wait );
01561 #endif
01562         (void) x;
01563         return wait( args... );
01564     }
01565
01591     template<
01592         Backend backend,
01593         typename InputType, typename RIT, typename CIT, typename NIT,
01594         typename... Args
01595     >
01596     RC wait(
01597         const Matrix< InputType, backend, RIT, CIT, NIT > &A,
01598         const Args &... args
01599     ) {
01600 #ifndef NDEBUG
01601         const bool should_not_call_base_matrix_wait = false;
01602         assert( should_not_call_base_matrix_wait );
01603 #endif
01604         (void) A;
01605         return wait( args... );
01606     }
01607
01610 } // namespace grb
01611
01612 #endif // end _H_GRB_IO_BASE
01613
```

10.53 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.

10.53.1 Detailed Description

Specifies the ALP/GraphBLAS matrix container.

Author

A. N. Yzelman

Date

10th of August, 2016

10.54 matrix.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_MATRIX_BASE
00028 #define _H_GRB_MATRIX_BASE
00029
00030 #include <iterator>
00031
00032 #include <stddef.h>
00033
00034 #include <utility>
00035
00036 #include <graphblas/backends.hpp>
00037 #include <graphblas/descriptors.hpp>
00038 #include <graphblas/ops.hpp>
00039 #include <graphblas/rc.hpp>
00040
00041
00042 namespace grb {
00043
00065     template<
00066         typename D, enum Backend implementation,
00067         typename RowIndexType,
00068         typename ColIndexType,
00069         typename NonzeroIndexType
00070     >
00071     class Matrix {
00072
00073         public :
00074
00076             typedef Matrix<
00077                 D, implementation,
00078                 RowIndexType, ColIndexType, NonzeroIndexType
00079             > self_type;
00080
00116             class const_iterator : public std::iterator<
00117                 std::forward_iterator_tag,
00118                 std::pair< std::pair< const size_t, const size_t >, const D >,
00119                 size_t
00120             > {
00121
00122                 public :
00123
00130                     bool operator==( const const_iterator &other ) const {
00131                         (void)other;
```

```
00132                         return false;
00133                     }
00134
00138                     bool operator!=( const const_iterator &other ) const {
00139                         (void)other;
00140                         return true;
00141                     }
00142
00155                     std::pair< const size_t, const D > operator*() const {
00156                         return std::pair< const size_t, const D >();
00157                     }
00158
00172                     const_iterator & operator++() {
00173                         return *this;
00174                     }
00175
00176             };
00177
00179             typedef D value_type;
00180
00205             Matrix( const size_t rows, const size_t columns, const size_t nz ) {
00206                 (void) rows;
00207                 (void) columns;
00208                 (void) nz;
00209             }
00210
00226             Matrix( const size_t rows, const size_t columns ) {
00227                 (void)rows;
00228                 (void)columns;
00229             }
00230
00260             Matrix(
00261                 const Matrix<
00262                     D, implementation,
00263                     RowIndexType, ColIndexType, NonzeroIndexType
00264                 > &other
00265             ) {
00266                 (void) other;
00267             }
00268
00283             Matrix( self_type &&other ) {
00284                 (void) other;
00285             }
00286
00303             self_type& operator=( self_type &&other ) noexcept {
00304                 *this = std::move( other );
00305                 return *this;
00306             }
00307
00321             ~Matrix() {}
00322
00349             const_iterator cbegin() const {}
00350
00358             const_iterator begin() const {}
00359
00380             const_iterator cend() const {}
00381
00389             const_iterator end() const {}
00390
00391     };
00392
00393 } // end namespace "grb"
00394
00395 #endif // end _H_GRB_MATRIX_BASE
00396
```

10.55 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.

10.55.1 Detailed Description

Contains the specification for grb::PinnedVector.

Author

A. N. Yzelman

10.56 pinnedvector.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00026 #ifndef _H_GRB_BASE_PINNEDVECTOR
00027 #define _H_GRB_BASE_PINNEDVECTOR
00028
00029 #include <limits>
00030
00031 #include <graphblas/backends.hpp>
00032 #include <graphblas/iomode.hpp>
00033
00034 #include "vector.hpp"
00035
00036
00037 namespace grb {
00038
00074     template< typename IOType, enum Backend implementation >
00075     class PinnedVector {
00076
00077         private :
00078
00082             static const constexpr bool
00083                 function_was_not_implemented_in_the_selected_backend = false;
00084
00085
00086         public :
00087
00133             template< typename Coord >
00134             PinnedVector(
00135                 const Vector< IOType, implementation, Coord > &vector,
00136                 const IOMode mode
00137             ) {
00138                 (void) vector;
00139                 (void) mode;
00140                 assert( function_was_not_implemented_in_the_selected_backend );
00141             }
00142
00153             PinnedVector() {
00154                 assert( function_was_not_implemented_in_the_selected_backend );
00155             }
00156
00166             ~PinnedVector() {
```

```
00167                 assert( function_was_not_implemented_in_the_selected_backend );
00168             }
00169
00180             inline size_t size() const noexcept {
00181                 assert( function_was_not_implemented_in_the_selected_backend );
00182                 return 0;
00183             }
00184
00195             inline size_t nonzeroes() const noexcept {
00196                 assert( function_was_not_implemented_in_the_selected_backend );
00197                 return 0;
00198             }
00199
00232             template< typename OutputType >
00233             inline OutputType getNonzeroValue(
00234                 const size_t k, const OutputType one = OutputType()
00235             ) const noexcept {
00236                 (void) k;
00237                 assert( function_was_not_implemented_in_the_selected_backend );
00238                 return one;
00239             }
00240
00256             inline IOType getNonzeroValue(
00257                 const size_t k
00258             ) const noexcept {
00259                 IOType ret;
00260                 (void) k;
00261                 assert( function_was_not_implemented_in_the_selected_backend );
00262                 return ret;
00263             }
00264
00282             inline size_t getNonzeroIndex(
00283                 const size_t k
00284             ) const noexcept {
00285                 (void) k;
00286                 assert( function_was_not_implemented_in_the_selected_backend );
00287                 return std::numeric_limits< size_t >::max();
00288             }
00289
00290
00291     }; // end class grb::PinnedVector
00292
00293 } // end namespace "grb"
00294
00295 #endif // end _H_GRB_BASE_PINNEDVECTOR
00296
```

10.57 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.

10.57.1 Detailed Description

Exposes facilities for direct SPMD programming.

Author

A. N. Yzelman

Date

28th of April, 2017

10.58 spmd.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_BASE_SPMD
00028 #define _H_GRB_BASE_SPMD
00029
00030 #include <cstddef> //size_t
00031
00032 #include <stdint.h> // SIZE_MAX
00033
00034 #include <graphblas/backends.hpp>
00035 #include <graphblas/rc.hpp>
00036
00037 #include "config.hpp"
00038
00039
00040 namespace grb {
00041
00050     template< Backend implementation >
00051     class spmd {
00052
00053         public:
00054
00056             static inline size_t nprocs() noexcept {
00057                 return 0;
00058             }
00059
00061             static inline size_t pid() noexcept {
00062                 return SIZE_MAX;
00063             }
00064
00091             static enum RC sync( const size_t msgs_in = 0, const size_t msgs_out = 0 ) noexcept {
00092                 (void) msgs_in;
00093                 (void) msgs_out;
00094                 return PANIC;
00095             }
00096
00097     }; // end class “spmd”
00098
00099 } // namespace grb
00100
00101 #endif // end _H_GRB_BASE_SPMD
00102
```

10.59 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.

10.59.1 Detailed Description

Specifies the ALP/GraphBLAS vector container.

Author

A. N. Yzelman

Date

10th of August, 2016

10.60 vector.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_VECTOR_BASE
00028 #define _H_GRB_VECTOR_BASE
00029
00030 #include <cstdlib>  //size_t
00031 #include <iterator> //std::iterator
00032 #include <stdexcept>
00033 #include <utility> //pair
00034
00035 #include <graphblas/backends.hpp>
00036 #include <graphblas/descriptors.hpp>
```

```
00037 #include <graphblas/ops.hpp>
00038 #include <graphblas/rc.hpp>
00039
00040
00041 namespace grb {
00042
00063     template< typename D, enum Backend implementation, typename C >
00064     class Vector {
00065
00066         public :
00067
00069             typedef D value_type;
00070
00124             typedef D& lambda_reference;
00125
00140             class const_iterator :
00141                 public std::iterator<
00142                     std::forward_iterator_tag,
00143                     std::pair< const size_t, const D >,
00144                     size_t
00145                 >
00146             {
00147
00148                 public :
00149
00151                     bool operator==( const const_iterator &other ) const {
00152                         (void) other;
00153                         return false;
00154                     }
00155
00157                     bool operator!=( const const_iterator &other ) const {
00158                         (void) other;
00159                         return true;
00160                     }
00161
00173                     std::pair< const size_t, const D > operator*() const {
00174                         return std::pair< const size_t, const D >();
00175                     }
00176
00190                     const_iterator & operator++() {
00191                         return *this;
00192                     }
00193
00194             };
00195
00226             Vector( const size_t n, const size_t nz ) {
00227                 (void) n;
00228                 (void) nz;
00229             }
00230
00235             Vector( const size_t n ) {
00236                 (void) n;
00237             }
00238
00259             Vector( Vector< D, implementation, C > &&x ) noexcept {
00260                 (void) x;
00261             }
00262
00278             Vector< D, implementation, C >& operator=(
00279                 Vector< D, implementation, C > &&x
00280             ) noexcept {
00281                 (void) x;
00282                 return *this;
00283             }
00284
00307             ~Vector() {}
00308
00310
00340             const_iterator cbegin() const {
00341                 const_iterator ret;
00342                 return ret;
00343             }
00344
00351             const_iterator begin() const {
00352                 const_iterator ret;
00353                 return ret;
00354             }
00355
00357
00359
00380             const_iterator cend() const {
00381                 const_iterator ret;
00382                 return ret;
00383             }
00384
00391             const_iterator end() const {
00392                 const_iterator ret;
```

```
00393                 return ret;
00394             }
00396
00480             template<
00481                 Descriptor descr = descriptors::no_operation,
00482                 class Accum = typename operators::right_assign< D, D, D >,
00483                 typename fwd_iterator = const D * __restrict__
00484             >
00485             RC build(
00486                 const Accum &accum,
00487                 const fwd_iterator start, const fwd_iterator end,
00488                 fwd_iterator npos
00489             ) {
00490                 (void) accum;
00491                 (void) start;
00492                 (void) end;
00493                 (void) npos;
00494                 return PANIC;
00495             }
00496
00595             template<
00596                 Descriptor descr = descriptors::no_operation,
00597                 class Accum = operators::right_assign< D, D, D >,
00598                 typename ind_iterator = const size_t * __restrict__,
00599                 typename nnz_iterator = const D * __restrict__,
00600                 class Dup = operators::right_assign< D, D, D >
00601             >
00602             RC build(
00603                 const Accum &accum,
00604                 const ind_iterator ind_start, const ind_iterator ind_end,
00605                 const nnz_iterator nnz_start, const nnz_iterator nnz_end,
00606                 const Dup &dup = Dup()
00607             ) {
00608                 (void) accum;
00609                 (void) ind_start;
00610                 (void) ind_end;
00611                 (void) nnz_start;
00612                 (void) nnz_end;
00613                 (void) dup;
00614                 return PANIC;
00615             }
00616
00720             template<
00721                 Descriptor descr = descriptors::no_operation,
00722                 typename mask_type,
00723                 class Accum,
00724                 typename ind_iterator = const size_t * __restrict__,
00725                 typename nnz_iterator = const D * __restrict__,
00726                 class Dup = operators::right_assign< D, typename nnz_iterator::value_type, D >
00727             >
00728             RC build(
00729                 const Vector< mask_type, implementation, C > &mask,
00730                 const Accum &accum,
00731                 const ind_iterator ind_start,
00732                 const ind_iterator ind_end,
00733                 const nnz_iterator nnz_start,
00734                 const nnz_iterator nnz_end,
00735                 const Dup &dup = Dup()
00736             ) {
00737                 (void) mask;
00738                 (void) accum;
00739                 (void) ind_start;
00740                 (void) ind_end;
00741                 (void) nnz_start;
00742                 (void) nnz_end;
00743                 (void) dup;
00744                 return PANIC;
00745             }
00746
00771             template< typename T >
00772             RC size( T &size ) const {
00773                 (void) size;
00774                 return PANIC;
00775             }
00776
00801             template< typename T >
00802             RC nnz( T &nnz ) const {
00803                 (void) nnz;
00804                 return PANIC;
00805             }
00806
00866             template< class Monoid >
00867             lambda_reference operator()(
00868                 const size_t i, const Monoid &monoid = Monoid()
00869             ) {
00870                 (void) i;
00871                 (void) monoid;
```

```
00872                 return PANIC;
00873             }
00874
00919             lambda_reference operator[]( const size_t i ) {
00920                 (void) i;
00921 #ifndef _GRB_NO_EXCEPTIONS
00922                 throw std::runtime_error(
00923                     "Requesting lambda reference of unimplemented Vector backend."
00924                 );
00925 #endif
00926             }
00927     };
00928
00929 } // end namespace "grb"
00930
00931 #endif // _H_GRB_VECTOR_BASE
00932
```

10.61 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.

10.61.1 Detailed Description

Defines the ALP/GraphBLAS level-0 API.

Author

A. N. Yzelman

Date

5th of December 2016

10.62 blas0.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_BLAS0
00028 #define _H_GRB_BLAS0
00029
00030 #include <functional>
00031 #include <stdexcept>
00032 #include <type_traits> //enable_if
00033
00034 #include "graphblas/descriptors.hpp"
00035 #include "graphblas/rc.hpp"
00036 #include "graphblas/type_traits.hpp"
00037
00038 #define NO_CAST_ASSERT( x, y, z )                                                \
00039     static_assert( x,                                                            \
00040         "\n\n"                                                                   \
00041         "******************************************************************" \
00042         "******************************************************************" \
00043         "**********************\n"                                              \
00044         "*     ERROR      | " y " " z ".\n"                                     \
00045         "******************************************************************" \
00046         "******************************************************************" \
00047         "**********************\n"                                              \
00048         "* Possible fix 1 | Remove no_casting from the template parameters in "  \
00049         "this call to " y ".\n"                                                  \
00050         "* Possible fix 2 | Provide a left-hand side input value of the same "   \
00051         "type as the first domain of the given operator.\n"                      \
00052         "* Possible fix 3 | Provide a right-hand side input value of the same "  \
00053         "type as the second domain of the given operator.\n"                     \
00054         "* Possible fix 4 | Provide an output value of the same type as the "    \
00055         "third domain of the given operator.\n"                                  \
00056         "* Note that in case of in-place operators the left-hand side input or "  \
00057         "right-hand side input also play the role of the output value.\n"        \
00058         "******************************************************************" \
00059         "******************************************************************" \
00060         "**********************\n" );
00061
00062
00063 namespace grb {
00064
00174     template<
00175         Descriptor descr = descriptors::no_operation,
00176         class OP,
00177         typename InputType1, typename InputType2, typename OutputType
00178     >
00179     static enum RC apply(
00180         OutputType &out,
00181         const InputType1 &x,
00182         const InputType2 &y,
00183         const OP &op = OP(),
00184         const typename std::enable_if<
00185             grb::is_operator< OP >::value &&
00186             !grb::is_object< InputType1 >::value &&
00187             !grb::is_object< InputType2 >::value &&
00188             !grb::is_object< OutputType >::value,
00189         void >::type * = nullptr
00190     ) {
00191         // static sanity check
00192         NO_CAST_ASSERT( ( !( descr & descriptors::no_casting ) || (
00193                 std::is_same< InputType1, typename OP::D1 >::value &&
00194                 std::is_same< InputType2, typename OP::D2 >::value &&
00195                 std::is_same< OutputType, typename OP::D3 >::value
00196             ) ),
00197             "grb::apply (BLAS level 0)",
00198             "Argument value types do not match operator domains while no_casting "
00199             "descriptor was set"
00200         );
```

```
00201
00202         // call apply
00203         const typename OP::D1 left = static_cast< typename OP::D1 >( x );
00204         const typename OP::D2 right = static_cast< typename OP::D2 >( y );
00205         typename OP::D3 output = static_cast< typename OP::D3 >( out );
00206         op.apply( left, right, output );
00207         out = static_cast< OutputType >( output );
00208
00209         // done
00210         return SUCCESS;
00211     }
00212
00283     template<
00284         Descriptor descr = descriptors::no_operation,
00285         class OP, typename InputType, typename IOType
00286     >
00287     static RC foldr(
00288         const InputType &x,
00289         IOType &y,
00290         const OP &op = OP(),
00291         const typename std::enable_if<
00292             grb::is_operator< OP >::value &&
00293             !grb::is_object< InputType >::value &&
00294             !grb::is_object< IOType >::value, void
00295         >::type * = nullptr
00296     ) {
00297         // static sanity check
00298         NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) || (
00299                 std::is_same< InputType, typename OP::D1 >::value &&
00300                 std::is_same< IOType, typename OP::D2 >::value &&
00301                 std::is_same< IOType, typename OP::D3 >::value
00302             ) ), "grb::foldr (BLAS level 0)",
00303             "Argument value types do not match operator domains while no_casting "
00304             "descriptor was set" );
00305
00306         // call foldr
00307         const typename OP::D1 left = static_cast< typename OP::D1 >( x );
00308         typename OP::D3 right = static_cast< typename OP::D3 >( y );
00309         op.foldr( left, right );
00310         y = static_cast< IOType >( right );
00311
00312         // done
00313         return SUCCESS;
00314     }
00315
00386     template<
00387         Descriptor descr = descriptors::no_operation,
00388         class OP,
00389         typename InputType, typename IOType
00390     >
00391     static RC foldl(
00392         IOType &x,
00393         const InputType &y,
00394         const OP &op = OP(),
00395         const typename std::enable_if< grb::is_operator< OP >::value &&
00396             !grb::is_object< InputType >::value &&
00397             !grb::is_object< IOType >::value, void
00398         >::type * = nullptr
00399     ) {
00400         // static sanity check
00401         NO_CAST_ASSERT( ( !(descr & descriptors::no_casting) || (
00402                 std::is_same< IOType, typename OP::D1 >::value &&
00403                 std::is_same< InputType, typename OP::D2 >::value &&
00404                 std::is_same< IOType, typename OP::D3 >::value
00405             ) ), "grb::foldl (BLAS level 0)",
00406             "Argument value types do not match operator domains while no_casting "
00407             "descriptor was set" );
00408
00409         // call foldl
00410         typename OP::D1 left = static_cast< typename OP::D1 >( x );
00411         const typename OP::D3 right = static_cast< typename OP::D3 >( y );
00412         op.foldl( left, right );
00413         x = static_cast< IOType >( left );
00414
00415         // done
00416         return SUCCESS;
00417     }
00418
00421     namespace internal {
00422
00437         template<
00438             grb::Descriptor descr,
00439             typename OutputType, typename D,
00440             typename Enabled = void
00441         >
00442         class ValueOrIndex;
00443
```

```
00444         /* Version where use_index is allowed. */
00445         template< grb::Descriptor descr, typename OutputType, typename D >
00446         class ValueOrIndex<
00447             descr,
00448             OutputType, D,
00449             typename std::enable_if<
00450                 std::is_arithmetic< OutputType >::value &&
00451                 !std::is_same< D, void >::value
00452             >::type
00453         > {
00454
00455             private:
00456
00457                 static constexpr const bool use_index = descr & grb::descriptors::use_index;
00458
00459                 static_assert( use_index || std::is_convertible< D, OutputType >::value,
00460                     "Cannot convert to the requested output type" );
00461
00462
00463             public:
00464
00465                 static OutputType getFromArray(
00466                     const D * __restrict__ const x,
00467                     const std::function< size_t( size_t ) > &src_local_to_global,
00468                     const size_t index
00469                 ) noexcept {
00470                     if( use_index ) {
00471                         return static_cast< OutputType >( src_local_to_global( index ) );
00472                     } else {
00473                         return static_cast< OutputType >( x[ index ] );
00474                     }
00475                 }
00476
00477                 static OutputType getFromScalar( const D &x, const size_t index ) noexcept {
00478                     if( use_index ) {
00479                         return static_cast< OutputType >( index );
00480                     } else {
00481                         return static_cast< OutputType >( x );
00482                     }
00483                 }
00484
00485         };
00486
00487         /* Version where use_index is not allowed. */
00488         template< grb::Descriptor descr, typename OutputType, typename D >
00489         class ValueOrIndex<
00490             descr,
00491             OutputType, D,
00492             typename std::enable_if<
00493                 !std::is_arithmetic< OutputType >::value &&
00494                 !std::is_same< OutputType, void >::value
00495             >::type
00496         > {
00497
00498             static_assert( !(descr & descriptors::use_index),
00499                 "use_index descriptor given while output type is not numeric" );
00500
00501             static_assert( std::is_convertible< D, OutputType >::value,
00502                 "Cannot convert input to the given output type" );
00503
00504             public:
00505
00506                 static OutputType getFromArray(
00507                     const D * __restrict__ const x,
00508                     const std::function< size_t( size_t ) > &,
00509                     const size_t index
00510                 ) noexcept {
00511                     return static_cast< OutputType >( x[ index ] );
00512                 }
00513
00514                 static OutputType getFromScalar(
00515                     const D &x, const size_t
00516                 ) noexcept {
00517                     return static_cast< OutputType >( x );
00518                 }
00519
00520         };
00521
00542         template<
00543             bool identity_left,
00544             typename OutputType, typename InputType,
00545             template< typename > class Identity,
00546             typename Enabled = void
00547         >
00548         class CopyOrApplyWithIdentity;
00549
00550         /* The cast-and-assign version */
```

```
00551         template<
00552             bool identity_left,
00553             typename OutputType, typename InputType,
00554             template< typename > class Identity
00555         >
00556         class CopyOrApplyWithIdentity<
00557             identity_left,
00558             OutputType, InputType,
00559             Identity,
00560             typename std::enable_if<
00561                 std::is_convertible< InputType, OutputType >::value
00562             >::type
00563         > {
00564
00565             public:
00566
00567                 template< typename Operator >
00568                 static void set( OutputType &out, const InputType &in, const Operator & ) {
00569                     out = static_cast< OutputType >( in );
00570                 }
00571
00572         };
00573
00574         /* The operator with identity version */
00575         template<
00576             bool identity_left,
00577             typename OutputType, typename InputType,
00578             template< typename > class Identity
00579         >
00580         class CopyOrApplyWithIdentity<
00581             identity_left,
00582             OutputType, InputType,
00583             Identity,
00584             typename std::enable_if<
00585                 !std::is_convertible< InputType, OutputType >::value
00586             >::type
00587         > {
00588
00589             public:
00590
00591                 template< typename Operator >
00592                 static void set(
00593                     OutputType &out, const InputType &in, const Operator &op
00594                 ) {
00595                     const auto identity = identity_left ?
00596                         Identity< typename Operator::D1 >::value() :
00597                         Identity< typename Operator::D2 >::value();
00598                     if( identity_left ) {
00599                         (void) grb::apply( out, identity, in, op );
00600                     } else {
00601                         (void) grb::apply( out, in, identity, op );
00602                     }
00603                 }
00604
00605         };
00606
00607     } // namespace internal
00608
00609 } // namespace grb
00610
00611 #undef NO_CAST_ASSERT
00612
00613 #endif // end "_H_GRB_BLAS0"
00614
```

10.63 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.

10.63.1 Detailed Description

Defines all ALP/GraphBLAS descriptors.

Author

A. N. Yzelman

Date

15 March, 2016

10.64 descriptors.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_DESCRIPTOR
00028 #define _H_GRB_DESCRIPTOR
00029
00030 #include <string>
00031
00032
00033 namespace grb {
00034
00054     typedef unsigned int Descriptor;
00055
00057     namespace descriptors {
00058
00063         static constexpr Descriptor no_operation = 0;
00064
00066         static constexpr Descriptor invert_mask = 1;
00067
00071         static constexpr Descriptor transpose_matrix = 2;
00072
00091         static constexpr Descriptor no_duplicates = 4;
00092
00103         static constexpr Descriptor structural = 8;
00104
00117         static constexpr Descriptor structural_complement = structural | invert_mask;
00118
00151         static constexpr Descriptor dense = 16;
00152
00159         static constexpr Descriptor add_identity = 32;
00160
00167         static constexpr Descriptor use_index = 64;
00168
00196         static constexpr Descriptor no_casting = 256;
00197
00207         static constexpr Descriptor explicit_zero = 512;
00208
00213         static constexpr Descriptor safe_overlap = 1024;
00214
00219         static constexpr Descriptor transpose_left = 2048;
00220
00225         static constexpr Descriptor transpose_right = 4096;
00226
00227         // Put internal, backend-specific descriptors last
00228
00229
00239         static constexpr Descriptor force_row_major = 8192;
00240
00248         std::string toString( const Descriptor descr );
00249
00250     } // namespace descriptors
00251
00252     namespace internal {
00253
00255         static constexpr Descriptor MAX_DESCRIPTOR_VALUE = 16383;
00256
00257     } // namespace internal
00258
00259 } // namespace grb
00260
00261 #endif
00262
```

10.65 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.

10.65.1 Detailed Description

Provides a set of standard identities for use with ALP.

Author

A. N. Yzelman

Date

11th of August, 2016

10.66 identities.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_IDENTITIES
00028 #define _H_GRB_IDENTITIES
00029
00030 #include <limits>
00031
00032
00033 namespace grb {
00034
00053     namespace identities {
00054
00056         template< typename D >
00057         class zero {
00058             static_assert( std::is_convertible< int, D >::value, "Cannot form identity under the
      requested domain" );
00059
00060         public:
00065             static constexpr D value() {
00066                 return static_cast< D >( 0 );
00067             }
00068         };
00069         template< typename K, typename V >
00070         class zero< std::pair< K, V > > {
00071         public:
00072             static constexpr std::pair< K, V > value() {
00073                 return std::make_pair( zero< K >::value(), zero< V >::value() );
00074             }
00075         };
00076
00078         template< typename D >
00079         class one {
00080             static_assert( std::is_convertible< int, D >::value, "Cannot form identity under the
      requested domain" );
00081
00082         public:
00087             static constexpr D value() {
00088                 return static_cast< D >( 1 );
00089             }
00090         };
00091         template< typename K, typename V >
00092         class one< std::pair< K, V > > {
00093         public:
00094             static constexpr std::pair< K, V > value() {
00095                 return std::make_pair( one< K >::value(), one< V >::value() );
00096             }
00097         };
00098
00100         template< typename D >
00101         class infinity {
00102             static_assert( std::is_arithmetic< D >::value, "Cannot form identity under the requested
      domain" );
00103
00104         public:
00110             static constexpr D value() {
00111                 return std::numeric_limits< D >::has_infinity ? std::numeric_limits< D >::infinity() :
      std::numeric_limits< D >::max();
00112             }
00113         };
00114         template< typename K, typename V >
00115         class infinity< std::pair< K, V > > {
00116         public:
00117             static constexpr std::pair< K, V > value() {
00118                 return std::make_pair( infinity< K >::value(), infinity< V >::value() );
00119             }
00120         };
00121
```

```
00123         template< typename D >
00124         class negative_infinity {
00125             static_assert( std::is_arithmetic< D >::value, "Cannot form identity under the requested
      domain" );
00126
00127         public:
00133             static constexpr D value() {
00134                 return std::numeric_limits< D >::min() == 0 ? 0 : ( std::numeric_limits< D
      >::has_infinity ? -std::numeric_limits< D >::infinity() : std::numeric_limits< D >::min() );
00135             }
00136         };
00137         template< typename K, typename V >
00138         class negative_infinity< std::pair< K, V > > {
00139         public:
00140             static constexpr std::pair< K, V > value() {
00141                 return std::make_pair( negative_infinity< K >::value(),
      negative_infinity< V >::value() );
00142             }
00143         };
00144
00150         template< typename D >
00151         class logical_false {
00152             static_assert( std::is_convertible< bool, D >::value, "Cannot form identity under the
      requested domain" );
00153
00154         public:
00160             static const constexpr D value() {
00161                 return static_cast< D >( false );
00162             }
00163         };
00164         template< typename K, typename V >
00165         class logical_false< std::pair< K, V > > {
00166         public:
00167             static constexpr std::pair< K, V > value() {
00168                 return std::make_pair( logical_false< K >::value(), logical_false< V >::value() );
00169             }
00170         };
00171
00177         template< typename D >
00178         class logical_true {
00179             static_assert( std::is_convertible< bool, D >::value, "Cannot form identity under the
      requested domain" );
00180
00181         public:
00187             static constexpr D value() {
00188                 return static_cast< D >( true );
00189             }
00190         };
00191         template< typename K, typename V >
00192         class logical_true< std::pair< K, V > > {
00193         public:
00194             static constexpr std::pair< K, V > value() {
00195                 return std::make_pair( logical_true< K >::value(), logical_true< V >::value() );
00196             }
00197         };
00198
00199     } // namespace identities
00200 } // namespace grb
00201
00202 #endif
```

10.67 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.

10.67.1 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

10.68 pregel.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00142 #ifndef _H_GRB_INTERFACES_PREGEL
00143 #define _H_GRB_INTERFACES_PREGEL
00144
00145 #include <graphblas.hpp>
```

```
00146 #include <graphblas/utils/parser.hpp>
00147
00148 #include <stdexcept> // std::runtime_error
00149
00150
00151 namespace grb {
00152
00153     namespace interfaces {
00154
00159         namespace config {
00160
00167             enum SparsificationStrategy {
00168
00174                 NONE = 0,
00175
00189                 ALWAYS,
00190
00209                 WHEN_REDUCED,
00210
00227                 WHEN_HALVED
00228
00229             };
00230
00242             constexpr const SparsificationStrategy out_sparsify = NONE;
00243
00244         } // end namespace grb::interfaces::config
00245
00266         struct PregelState {
00267
00282             bool &active;
00283
00291             bool &voteToHalt;
00292
00296             const size_t &num_vertices;
00297
00301             const size_t &num_edges;
00302
00306             const size_t &outdegree;
00307
00311             const size_t &indegree;
00312
00316             const size_t &round;
00317
00324             const size_t &vertexID;
00325
00326         };
00327
00337         template<
00338             typename MatrixEntryType
00339         >
00340         class Pregel {
00341
00342             private:
00343
00345                 const size_t n;
00346
00348                 size_t nz;
00349
00351                 grb::Matrix< MatrixEntryType > graph;
00352
00354                 grb::Vector< bool > activeVertices;
00355
00357                 grb::Vector< bool > haltVotes;
00358
00360                 grb::Vector< bool > buffer;
00361
00363                 grb::Vector< size_t > outdegrees;
00364
00366                 grb::Vector< size_t > indegrees;
00367
00369                 grb::Vector< size_t > IDs;
00370
00380                 void initialize() {
00381                     grb::Semiring<
00382                         grb::operators::add< size_t >,
00383                         grb::operators::right_assign_if< bool, size_t, size_t >,
00384                         grb::identities::zero,
00385                         grb::identities::logical_true
00386                     > ring;
00387                     grb::Vector< size_t > ones( n );
00388                     if( grb::set( ones, 1 ) != SUCCESS ) {
00389                         throw std::runtime_error( "Could not set vector ones" );
00390                     }
00391                     if( grb::set( outdegrees, 0 ) != SUCCESS ) {
00392                         throw std::runtime_error( "Could not initialise outdegrees" );
00393                     }
00394                     if( grb::mxv< grb::descriptors::dense >(
```

```
00395                             outdegrees, graph, ones, ring
00396                         ) != SUCCESS
00397                     ) {
00398                         throw std::runtime_error( "Could not compute outdegrees" );
00399                     }
00400                     if( grb::set( indegrees, 0 ) != SUCCESS ) {
00401                         throw std::runtime_error( "Could not initialise indegrees" );
00402                     }
00403                     if( grb::mxv<
00404                         grb::descriptors::dense | grb::descriptors::transpose_matrix
00405                     >(
00406                         indegrees, graph, ones, ring
00407                     ) != SUCCESS ) {
00408                         throw std::runtime_error( "Could not compute indegrees" );
00409                     }
00410                     if( grb::set< grb::descriptors::use_index >(
00411                             IDs, 0
00412                         ) != SUCCESS
00413                     ) {
00414                         throw std::runtime_error( "Could not compute vertex IDs" );
00415                     }
00416                 }
00417
00418
00419             protected:
00420
00427                 template< typename IType >
00428                 Pregel(
00429                     const size_t _n,
00430                     IType _start, const IType _end,
00431                     const grb::IOMode _mode
00432                 ) :
00433                     n( _n ),
00434                     graph( _n, _n ),
00435                     activeVertices( _n ),
00436                     haltVotes( _n ),
00437                     buffer( _n ),
00438                     outdegrees( _n ),
00439                     indegrees( _n ),
00440                     IDs( _n )
00441                 {
00442                     if( grb::ncols( graph ) != grb::nrows( graph ) ) {
00443                         throw std::runtime_error( "Input graph is bipartite" );
00444                     }
00445                     if( grb::buildMatrixUnique(
00446                         graph, _start, _end, _mode
00447                     ) != SUCCESS ) {
00448                         throw std::runtime_error( "Could not build graph" );
00449                     }
00450                     nz = grb::nnz( graph );
00451                     initialize();
00452                 }
00453
00454
00455             public:
00456
00500                 template< typename IType >
00501                 Pregel(
00502                     const size_t _m, const size_t _n,
00503                     IType _start, const IType _end,
00504                     const grb::IOMode _mode
00505                 ) : Pregel( std::max( _m, _n ), _start, _end, _mode ) {}
00506
00641                 template<
00642                     class Op,
00643                     template< typename > class Id,
00644                     class Program,
00645                     typename IOType,
00646                     typename GlobalProgramData,
00647                     typename IncomingMessageType,
00648                     typename OutgoingMessageType
00649                 >
00650                 grb::RC execute(
00651                     const Program program,
00652                     grb::Vector< IOType > &vertex_state,
00653                     const GlobalProgramData &data,
00654                     grb::Vector< IncomingMessageType > &in,
00655                     grb::Vector< OutgoingMessageType > &out,
00656                     size_t &rounds,
00657                     grb::Vector< OutgoingMessageType > &out_buffer =
00658                         grb::Vector< OutgoingMessageType >(0),
00659                     const size_t max_rounds = 0
00660                 ) {
00661                     static_assert( grb::is_operator< Op >::value &&
00662                             grb::is_associative< Op >::value,
00663                         "The combiner must be an associate operator"
00664                     );
```

```
00665                     static_assert( std::is_same< typename Op::D1, IncomingMessageType >::value,
00666                         "The combiner left-hand input domain should match the incoming message "
00667                         "type." );
00668                     static_assert( std::is_same< typename Op::D1, IncomingMessageType >::value,
00669                         "The combiner right-hand input domain should match the incoming message "
00670                         "type." );
00671                     static_assert( std::is_same< typename Op::D1, IncomingMessageType >::value,
00672                         "The combiner output domain should match the incoming message type." );
00673
00674                     // set default output
00675                     rounds = 0;
00676
00677                     // sanity checks
00678                     if( grb::size(vertex_state) != n ) {
00679                         return MISMATCH;
00680                     }
00681                     if( grb::size(in) != n ) {
00682                         return MISMATCH;
00683                     }
00684                     if( grb::size(out) != n ) {
00685                         return MISMATCH;
00686                     }
00687                     if( grb::capacity(vertex_state) != n ) {
00688                         return ILLEGAL;
00689                     }
00690                     if( grb::capacity(in) != n ) {
00691                         return ILLEGAL;
00692                     }
00693                     if( grb::capacity(out) != n ) {
00694                         return ILLEGAL;
00695                     }
00696                     if( config::out_sparsify && grb::capacity(out_buffer) != n ) {
00697                         return ILLEGAL;
00698                     }
00699                     if( grb::nnz(vertex_state) != n ) {
00700                         return ILLEGAL;
00701                     }
00702
00703                     // define some monoids and semirings
00704                     grb::Monoid<
00705                         grb::operators::logical_or< bool >,
00706                         grb::identities::logical_false
00707                     > orMonoid;
00708
00709                     grb::Monoid<
00710                         grb::operators::logical_and< bool >,
00711                         grb::identities::logical_true
00712                     > andMonoid;
00713
00714                     grb::Semiring<
00715                         Op,
00716                         grb::operators::left_assign_if<
00717                             IncomingMessageType, bool, IncomingMessageType
00718                         >,
00719                         Id,
00720                         grb::identities::logical_true
00721                     > ring;
00722
00723                     // set initial round ID
00724                     size_t step = 0;
00725
00726                     // activate all vertices
00727                     grb::RC ret = grb::set( activeVertices, true );
00728
00729                     // initialise halt votes to all-false
00730                     if( ret == SUCCESS ) {
00731                         ret = grb::set( haltVotes, false );
00732                     }
00733
00734                     // set default incoming message
00735                     if( ret == SUCCESS && grb::nnz(in) < n ) {
00736 #ifdef _DEBUG
00737                         if( grb::nnz(in) > 0 ) {
00738                             std::cerr « "Overwriting initial incoming messages since it was not a "
00739                                 « "dense vector\n";
00740                         }
00741 #endif
00742                         ret = grb::set( in, Id< IncomingMessageType >::value() );
00743                     }
00744
00745                     // reset outgoing buffer
00746                     size_t out_nnz = n;
00747                     if( ret == SUCCESS ) {
00748                         ret = grb::set( out, Id< OutgoingMessageType >::value() );
00749                     }
00750
00751                     // return if initialisation failed
```

```
00752                     if( ret != SUCCESS ) {
00753                         assert( ret == FAILED );
00754                         std::cerr « "Error: initialisation failed, but if workspace holds full "
00755                             « "capacity, initialisation should never fail. Please submit a bug "
00756                             « "report.\n";
00757                         return PANIC;
00758                     }
00759
00760                     // while there are active vertices, execute
00761                     while( ret == SUCCESS ) {
00762
00763                         assert( max_rounds == 0 || step < max_rounds );
00764                         // run one step of the program
00765                         ret = grb::eWiseLambda(
00766                             [
00767                                 this,
00768                                 &vertex_state,
00769                                 &in,
00770                                 &out,
00771                                 &program,
00772                                 &step,
00773                                 &data
00774                             ]( const size_t i ) {
00775                                 // create Pregel struct
00776                                 PregelState pregel = {
00777                                     activeVertices[ i ],
00778                                     haltVotes[ i ],
00779                                     n,
00780                                     nz,
00781                                     outdegrees[ i ],
00782                                     indegrees[ i ],
00783                                     step,
00784                                     IDs[ i ]
00785                                 };
00786                                 // only execute program on active vertices
00787                                 assert( activeVertices[ i ] );
00788 #ifdef _DEBUG
00789                                 std::cout « "Vertex " « i « " remains active in step " « step
00790                                     « "\n";
00791 #endif
00792                                 program(
00793                                     vertex_state[ i ],
00794                                     in[ i ],
00795                                     out[ i ],
00796                                     data,
00797                                     pregel
00798                                 );
00799 #ifdef _DEBUG
00800                                 std::cout « "Vertex " « i « " sends out message " « out[ i ]
00801                                     « "\n";
00802 #endif
00803                             }, activeVertices, vertex_state, in, out, outdegrees, haltVotes
00804                         );
00805
00806                         // increment counter
00807                         (void) ++step;
00808
00809                         // check if everyone voted to halt
00810                         if( ret == SUCCESS ) {
00811                             bool halt = true;
00812                             ret = grb::foldl< grb::descriptors::structural >(
00813                                 halt, haltVotes, activeVertices, andMonoid
00814                             );
00815                             assert( ret == SUCCESS );
00816                             if( ret == SUCCESS && halt ) {
00817 #ifdef _DEBUG
00818                                 std::cout « "\t All active vertices voted to halt; "
00819                                     « "terminating Pregel program.\n";
00820 #endif
00821                                 break;
00822                             }
00823                         }
00824
00825                         // update active vertices
00826                         if( ret == SUCCESS ) {
00827 #ifdef _DEBUG
00828                             std::cout « "\t Number of active vertices was "
00829                                 « grb::nnz( activeVertices ) « ", and ";
00830 #endif
00831                             ret = grb::clear( buffer );
00832                             ret = ret ? ret : grb::set( buffer, activeVertices, true );
00833                             std::swap( buffer, activeVertices );
00834 #ifdef _DEBUG
00835                             std::cout « " has now become " « grb::nnz( activeVertices ) « "\n";
00836 #endif
00837                         }
00838
```

```
00839                         // check if there is a next round
00840                         const size_t curActive = grb::nnz( activeVertices );
00841                         if( ret == SUCCESS && curActive == 0 ) {
00842 #ifdef _DEBUG
00843                             std::cout « "\t All vertices are inactive; "
00844                                 « "terminating Pregel program.\n";
00845 #endif
00846                             break;
00847                         }
00848
00849                         // check if we exceed the maximum number of rounds
00850                         if( max_rounds > 0 && step > max_rounds ) {
00851 #ifdef _DEBUG
00852                             std::cout « "\t Maximum number of Pregel rounds met "
00853                                 « "without the program returning a valid termination condition. "
00854                                 « "Exiting prematurely with a FAILED error code.\n";
00855 #endif
00856                             ret = FAILED;
00857                             break;
00858                         }
00859
00860 #ifdef _DEBUG
00861                         std::cout « "\t Starting message exchange\n";
00862 #endif
00863
00864                         // reset halt votes
00865                         if( ret == SUCCESS ) {
00866                             ret = grb::clear( haltVotes );
00867                             ret = ret ? ret : grb::set< grb::descriptors::structural >(
00868                                 haltVotes, activeVertices, false
00869                             );
00870                         }
00871
00872                         // reset incoming buffer
00873                         if( ret == SUCCESS ) {
00874                             ret = grb::clear( in );
00875                             ret = ret ? ret : grb::set< grb::descriptors::structural >(
00876                                 in, activeVertices, Id< IncomingMessageType >::value()
00877                             );
00878                         }
00879
00880                         // execute communication
00881                         if( ret == SUCCESS ) {
00882                             ret = grb::vxm< grb::descriptors::structural >(
00883                                 in, activeVertices, out, graph, ring
00884                             );
00885                         }
00886
00887                         // sparsify and reset outgoing buffer
00888                         if( config::out_sparsify && ret == SUCCESS ) {
00889                             if( config::out_sparsify == config::ALWAYS ||
00890                                 (config::out_sparsify == config::WHEN_REDUCED && out_nnz > curActive)
    ||
00891                                 (config::out_sparsify == config::WHEN_HALVED && curActive <=
    out_nnz/2)
00892                             ) {
00893                                 ret = grb::clear( out_buffer );
00894                                 ret = ret ? ret : grb::set< grb::descriptors::structural >(
00895                                         out_buffer, activeVertices, Id< OutgoingMessageType >::value()
00896                                     );
00897                                 std::swap( out, out_buffer );
00898                                 out_nnz = curActive;
00899                             }
00900                         }
00901
00902 #ifdef _DEBUG
00903                         std::cout « "\t Resetting outgoing message fields and "
00904                             « "starting next compute round\n";
00905 #endif
00906
00907                     }
00908
00909 #ifdef _DEBUG
00910                     if( grb::spmd<>::pid() == 0 ) {
00911                         std::cout « "Info: Pregel exits after " « step
00912                             « " rounds with error code " « ret
00913                             « " ( " « grb::toString(ret) « " )\n";
00914                     }
00915 #endif
00916
00917                     // done
00918                     rounds = step;
00919                     return ret;
00920                 }
00921
00928                 size_t num_vertices() const noexcept { return n; }
00929
```

```
00936                 size_t num_edges() const noexcept { return nz; }
00937
00949                 const grb::Matrix< MatrixEntryType > & get_matrix() const noexcept {
                          return graph;
00951                 }
00952
00953         };
00954
00955     } // end namespace "grb::interfaces"
00956
00957 } // end namespace "grb"
00958
00959 #endif // end "_H_GRB_INTERFACES_PREGEL"
00960
```

10.69 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.

10.69.1 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

10.70 iomode.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00028 #ifndef _H_GRB_IOMODE
00029 #define _H_GRB_IOMODE
00030
00031
00032 namespace grb {
00033
00067     enum IOMode {
00068
00075         SEQUENTIAL = 0,
00076
00092         PARALLEL
00093     };
00094
00095 } // namespace grb
00096
00097 #endif // end "_H_GRB_IOMODE"
00098
```

10.71 monoid.hpp File Reference

Provides an ALP monoid.

Classes

- class Monoid< _OP, _ID >

  A generalised monoid.

Namespaces

- namespace grb

  The ALP/GraphBLAS namespace.

10.71.1 Detailed Description

Provides an ALP monoid.

Author

A. N. Yzelman

Date

15 March, 2016

10.72 monoid.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_MONOID
00028 #define _H_GRB_MONOID
00029
00030 #ifdef _DEBUG
00031 #include <cstdio>
00032 #endif
00033
00034 #include <cstddef> //size_t
00035 #include <cstdlib> //posix_memalign, rand
00036 #include <type_traits>
00037
00038 #include <assert.h>
00039
00040 #include <graphblas/identities.hpp>
00041 #include <graphblas/ops.hpp>
00042 #include <graphblas/type_traits.hpp>
00043
00044
00045 namespace grb {
00046
00053     template< class _OP, template< typename > class _ID >
00054     class Monoid {
00055
00056         static_assert( grb::is_operator< _OP >::value, "First template argument to Monoid must be a
      GraphBLAS operator" );
00057
00058         static_assert( grb::is_associative< _OP >::value,
00059             "Cannot form a monoid using the given operator since it is not "
00060             "associative" );
00061
00062         static_assert( std::is_same< typename _OP::D1, typename _OP::D3 >::value || std::is_same<
      typename _OP::D2, typename _OP::D3 >::value,
00063             "Cannot form a monoid when the output domain does not match at least "
00064             "one of its input domains" );
00065
00066     public:
00067
00069         typedef typename _OP::D1 D1;
00070
00072         typedef typename _OP::D2 D2;
00073
00075         typedef typename _OP::D3 D3;
00076
00078         typedef _OP Operator;
00079
00081         template< typename IdentityType >
00082         using Identity = _ID< IdentityType >;
00083
00084
00085     private:
00086
00092         Operator op;
00093
00094     public:
00099         Monoid() : op() {}
00100
00110         template< typename D >
00111         constexpr D getIdentity() const {
00112             return Identity< D >::value();
00113         }
00114
00120         Operator getOperator() const {
00121             return op;
00122         }
00123     };
```

```
00124
00125     // type traits
00126     template< class _OP, template< typename > class _ID >
00127     struct is_monoid< Monoid< _OP, _ID > > {
00129         static const constexpr bool value = true;
00130     };
00131
00132     template< class OP, template< typename > class ID >
00133     struct has_immutable_nonzeroes< Monoid< OP, ID > > {
00134         static const constexpr bool value = grb::is_monoid< Monoid< OP, ID > >::value &&
00135             std::is_same< OP, typename grb::operators::logical_or< typename OP::D1, typename OP::D2,
      typename OP::D3 > >::value;
00136     };
00137
00138 } // namespace grb
00139
00140 #endif
```

10.73 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 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 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.

10.73.1 Detailed Description

Provides a set of standard binary operators.

Author

A. N. Yzelman

Date

8th of August, 2016

10.73.2 Macro Definition Documentation

10.73.2.1 _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.

10.74 ops.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_OPERATORS
00028 #define _H_GRB_OPERATORS
00029
00030 #include "internalops.hpp"
00031 #include "type_traits.hpp"
00032
00033
00034 namespace grb {
00035
00040     namespace operators {
00041
00054         template<
00055             typename D1, typename D2 = D1, typename D3 = D2,
00056             enum Backend implementation = config::default_backend
00057         >
00058         class left_assign :
00059             public internal::Operator<
00060                 internal::left_assign< D1, D2, D3, implementation >
00061             >
00062         {
00063             public:
00064
00065                 template< typename A, typename B, typename C, enum Backend D >
00066                 using GenericOperator = left_assign< A, B, C, D >;
00067
00068                 left_assign() {}
00069         };
00070
00080         template<
00081             typename D1, typename D2 = D1, typename D3 = D2,
00082             enum Backend implementation = config::default_backend
00083         >
00084         class left_assign_if :
00085              public internal::Operator<
00086                     internal::left_assign_if< D1, D2, D3, implementation >
00087             >
00088         {
00089             public:
00090
00091                 template< typename A, typename B, typename C, enum Backend D >
00092                 using GenericOperator = left_assign_if< A, B, C, D >;
00093
00094                 left_assign_if() {}
00095         };
00096
00109         template<
00110             typename D1, typename D2 = D1, typename D3 = D2,
00111             enum Backend implementation = config::default_backend
00112         >
```

```
00113         class right_assign : public internal::Operator<
00114                 internal::right_assign< D1, D2, D3, implementation >
00115             >
00116         {
00117             public:
00118 
00119                 template< typename A, typename B, typename C, enum Backend D >
00120                 using GenericOperator = right_assign< A, B, C, D >;
00121 
00122                 right_assign() {}
00123         };
00124 
00134         template<
00135             typename D1, typename D2 = D1, typename D3 = D2,
00136             enum Backend implementation = config::default_backend
00137         >
00138         class right_assign_if : public internal::Operator<
00139                 internal::right_assign_if< D1, D2, D3, implementation >
00140             >
00141         {
00142             public:
00143 
00144                 template< typename A, typename B, typename C, enum Backend D >
00145                 using GenericOperator = right_assign_if< A, B, C, D >;
00146 
00147                 right_assign_if() {}
00148         };
00149 
00166         // [Operator Wrapping]
00167         template<
00168             typename D1, typename D2 = D1, typename D3 = D2,
00169             enum Backend implementation = config::default_backend
00170         >
00171         class add : public internal::Operator<
00172                 internal::add< D1, D2, D3, implementation >
00173             >
00174         {
00175 
00176             public:
00177 
00178                 template< typename A, typename B, typename C, enum Backend D >
00179                 using GenericOperator = add< A, B, C, D >;
00180 
00181                 add() {}
00182         };
00183         // [Operator Wrapping]
00184 
00201         template<
00202             typename D1, typename D2 = D1, typename D3 = D2,
00203             enum Backend implementation = config::default_backend
00204         >
00205         class mul : public internal::Operator<
00206                 internal::mul< D1, D2, D3, implementation >
00207             >
00208         {
00209             public:
00210 
00211                 template< typename A, typename B, typename C, enum Backend D >
00212                 using GenericOperator = mul< A, B, C, D >;
00213 
00214                 mul() {}
00215         };
00216 
00234         template<
00235             typename D1, typename D2 = D1, typename D3 = D2,
00236             enum Backend implementation = config::default_backend
00237         >
00238         class max : public internal::Operator<
00239                 internal::max< D1, D2, D3, implementation >
00240             >
00241         {
00242             public:
00243 
00244                 template< typename A, typename B, typename C, enum Backend D >
00245                 using GenericOperator = max< A, B, C, D >;
00246 
00247                 max() {}
00248         };
00249 
00267         template<
00268             typename D1, typename D2 = D1, typename D3 = D2,
00269             enum Backend implementation = config::default_backend
00270         >
00271         class min : public internal::Operator<
00272                 internal::min< D1, D2, D3, implementation >
00273             >
00274         {
```

```
00275             public:
00276
00277                 template< typename A, typename B, typename C, enum Backend D >
00278                 using GenericOperator = min< A, B, C, D >;
00279
00280                 min() {}
00281         };
00282
00294         template<
00295             typename D1, typename D2 = D1, typename D3 = D2,
00296             enum Backend implementation = config::default_backend
00297         >
00298         class subtract : public internal::Operator<
00299                 internal::substract< D1, D2, D3, implementation >
00300             >
00301         {
00302             public:
00303
00304                 template< typename A, typename B, typename C, enum Backend D >
00305                 using GenericOperator = subtract< A, B, C, D >;
00306
00307                 subtract() {}
00308         };
00309
00321         template<
00322             typename D1, typename D2 = D1, typename D3 = D2,
00323             enum Backend implementation = config::default_backend
00324         >
00325         class divide : public internal::Operator<
00326                 internal::divide< D1, D2, D3, implementation >
00327             >
00328         {
00329             public:
00330
00331                 template< typename A, typename B, typename C, enum Backend D >
00332                 using GenericOperator = divide< A, B, C, D >;
00333
00334                 divide() {}
00335         };
00336
00346         template<
00347             typename D1, typename D2 = D1, typename D3 = D2,
00348             enum Backend implementation = config::default_backend
00349         >
00350         class divide_reverse : public internal::Operator<
00351                 internal::divide_reverse< D1, D2, D3, implementation >
00352             >
00353         {
00354             public:
00355
00356                 template< typename A, typename B, typename C, enum Backend D >
00357                 using GenericOperator = divide_reverse< A, B, C, D >;
00358
00359                 divide_reverse() {}
00360         };
00361
00372         template<
00373             typename D1, typename D2 = D1, typename D3 = D2,
00374             enum Backend implementation = config::default_backend
00375         >
00376         class equal : public internal::Operator<
00377                 internal::equal< D1, D2, D3, implementation >
00378             >
00379         {
00380             public:
00381
00382                 template< typename A, typename B, typename C, enum Backend D >
00383                 using GenericOperator = equal< A, B, C, D >;
00384
00385                 equal() {}
00386         };
00387
00398         template<
00399             typename D1, typename D2 = D1, typename D3 = D2,
00400             enum Backend implementation = config::default_backend
00401         >
00402         class not_equal : public internal::Operator<
00403                 internal::not_equal< D1, D2, D3, implementation >
00404             >
00405         {
00406             public:
00407
00408                 template< typename A, typename B, typename C, enum Backend D >
00409                 using GenericOperator = not_equal< A, B, C, D >;
00410
00411                 not_equal() {}
00412         };
```

```
00413
00427         template<
00428             typename D1, typename D2 = D1, typename D3 = D2,
00429             enum Backend implementation = config::default_backend
00430         >
00431         class any_or : public internal::Operator<
00432                 internal::any_or< D1, D2, D3, implementation >
00433             >
00434         {
00435             public:
00436
00437                 template< typename A, typename B, typename C, enum Backend D >
00438                 using GenericOperator = any_or< A, B, C, D >;
00439
00440                 any_or() {}
00441         };
00442
00455         template<
00456             typename D1, typename D2 = D1, typename D3 = D2,
00457             enum Backend implementation = config::default_backend
00458         >
00459         class logical_or : public internal::Operator<
00460                 internal::logical_or< D1, D2, D3, implementation >
00461             >
00462         {
00463
00464             public:
00465
00466                 template< typename A, typename B, typename C, enum Backend D >
00467                 using GenericOperator = logical_or< A, B, C, D >;
00468
00469                 logical_or() {}
00470         };
00471
00484         template<
00485             typename D1, typename D2 = D1, typename D3 = D2,
00486             enum Backend implementation = config::default_backend
00487         >
00488         class logical_and : public internal::Operator<
00489                 internal::logical_and< D1, D2, D3, implementation >
00490             >
00491         {
00492             public:
00493
00494                 template< typename A, typename B, typename C, enum Backend D >
00495                 using GenericOperator = logical_and< A, B, C, D >;
00496
00497                 logical_and() {}
00498         };
00499
00508         template<
00509             typename D1, typename D2 = D1, typename D3 = D2,
00510             enum Backend implementation = config::default_backend
00511         >
00512         class relu : public internal::Operator<
00513                 internal::relu< D1, D2, D3, implementation >
00514             >
00515         {
00516             public:
00517
00518                 template< typename A, typename B, typename C, enum Backend D >
00519                 using GenericOperator = relu< A, B, C, D >;
00520
00521                 relu() {}
00522         };
00523
00535         template<
00536             typename D1, typename D2 = D1, typename D3 = D2,
00537             enum Backend implementation = config::default_backend
00538         >
00539         class abs_diff : public internal::Operator<
00540                 internal::abs_diff< D1, D2, D3, implementation >
00541             >
00542         {
00543             public:
00544
00545                 template< typename A, typename B, typename C, enum Backend D >
00546                 using GenericOperator = abs_diff< A, B, C, D >;
00547
00548                 abs_diff() {}
00549         };
00550
00569         template< typename IType, typename VType >
00570         class argmin : public internal::Operator< internal::argmin< IType, VType > > {
00571             public:
00572                 argmin() {}
00573         };
```

```
00574
00593         template< typename IType, typename VType >
00594         class argmax : public internal::Operator< internal::argmax< IType, VType > > {
00595             public:
00596                 argmax() {}
00597         };
00598
00610         template<
00611             typename D1, typename D2, typename D3,
00612             enum Backend implementation = config::default_backend
00613         >
00614         class square_diff : public internal::Operator<
00615                 internal::square_diff< D1, D2, D3, implementation >
00616             >
00617         {
00618             public:
00619
00620                 template< typename A, typename B, typename C, enum Backend D >
00621                 using GenericOperator = square_diff< A, B, C, D >;
00622
00623                 square_diff() {}
00624         };
00625
00635         template<
00636             typename IN1, typename IN2,
00637             enum Backend implementation = config::default_backend
00638         >
00639         class zip : public internal::Operator<
00640                 internal::zip< IN1, IN2, implementation >
00641             >
00642         {
00643             public:
00644
00645                 template< typename A, typename B, enum Backend D >
00646                 using GenericOperator = zip< A, B, D >;
00647
00648                 zip() {}
00649         };
00650
00663         template<
00664             typename D1, typename D2 = D1, typename D3 = D2,
00665             enum Backend implementation = config::default_backend
00666         >
00667         class equal_first : public internal::Operator<
00668                 internal::equal_first< D1, D2, D3, implementation >
00669             >
00670         {
00671             public:
00672
00673                 template< typename A, typename B, typename C, enum Backend D >
00674                 using GenericOperator = equal_first< A, B, C, D >;
00675
00676                 equal_first() {}
00677         };
00678
00679     } // namespace operators
00680
00681     template< typename D1, typename D2, typename D3, enum Backend implementation >
00682     struct is_operator< operators::left_assign_if< D1, D2, D3, implementation > > {
00683         static const constexpr bool value = true;
00684     };
00685
00686     template< typename D1, typename D2, typename D3, enum Backend implementation >
00687     struct is_operator< operators::right_assign_if< D1, D2, D3, implementation > > {
00688         static const constexpr bool value = true;
00689     };
00690
00691     template< typename D1, typename D2, typename D3, enum Backend implementation >
00692     struct is_operator< operators::left_assign< D1, D2, D3, implementation > > {
00693         static const constexpr bool value = true;
00694     };
00695
00696     template< typename D1, typename D2, typename D3, enum Backend implementation >
00697     struct is_operator< operators::right_assign< D1, D2, D3, implementation > > {
00698         static const constexpr bool value = true;
00699     };
00700
00701     // [Operator Type Traits]
00702     template< typename D1, typename D2, typename D3, enum Backend implementation >
00703     struct is_operator< operators::add< D1, D2, D3, implementation > > {
00704         static const constexpr bool value = true;
00705     };
00706     // [Operator Type Traits]
00707
00708     template< typename D1, typename D2, typename D3, enum Backend implementation >
00709     struct is_operator< operators::mul< D1, D2, D3, implementation > > {
00710         static const constexpr bool value = true;
```

```
00711     };
00712
00713     template< typename D1, typename D2, typename D3, enum Backend implementation >
00714     struct is_operator< operators::max< D1, D2, D3, implementation > > {
00715         static const constexpr bool value = true;
00716     };
00717
00718     template< typename D1, typename D2, typename D3, enum Backend implementation >
00719     struct is_operator< operators::min< D1, D2, D3, implementation > > {
00720         static const constexpr bool value = true;
00721     };
00722
00723     template< typename D1, typename D2, typename D3, enum Backend implementation >
00724     struct is_operator< operators::subtract< D1, D2, D3, implementation > > {
00725         static const constexpr bool value = true;
00726     };
00727
00728     template< typename D1, typename D2, typename D3, enum Backend implementation >
00729     struct is_operator< operators::divide< D1, D2, D3, implementation > > {
00730         static const constexpr bool value = true;
00731     };
00732
00733     template< typename D1, typename D2, typename D3, enum Backend implementation >
00734     struct is_operator< operators::divide_reverse< D1, D2, D3, implementation > > {
00735         static const constexpr bool value = true;
00736     };
00737
00738     template< typename D1, typename D2, typename D3, enum Backend implementation >
00739     struct is_operator< operators::equal< D1, D2, D3, implementation > > {
00740         static const constexpr bool value = true;
00741     };
00742
00743     template< typename D1, typename D2, typename D3, enum Backend implementation >
00744     struct is_operator< operators::not_equal< D1, D2, D3, implementation > > {
00745         static const constexpr bool value = true;
00746     };
00747
00748     template< typename D1, typename D2, typename D3, enum Backend implementation >
00749     struct is_operator< operators::any_or< D1, D2, D3, implementation > > {
00750         static const constexpr bool value = true;
00751     };
00752
00753     template< typename D1, typename D2, typename D3, enum Backend implementation >
00754     struct is_operator< operators::logical_or< D1, D2, D3, implementation > > {
00755         static const constexpr bool value = true;
00756     };
00757
00758     template< typename D1, typename D2, typename D3, enum Backend implementation >
00759     struct is_operator< operators::logical_and< D1, D2, D3, implementation > > {
00760         static const constexpr bool value = true;
00761     };
00762
00763     template< typename D1, typename D2, typename D3, enum Backend implementation >
00764     struct is_operator< operators::abs_diff< D1, D2, D3, implementation > > {
00765         static const constexpr bool value = true;
00766     };
00767
00768     template< typename D1, typename D2, typename D3, enum Backend implementation >
00769     struct is_operator< operators::relu< D1, D2, D3, implementation > > {
00770         static const constexpr bool value = true;
00771     };
00772
00773     template< typename IType, typename VType >
00774     struct is_operator< operators::argmin< IType, VType > > {
00775         static const constexpr bool value = true;
00776     };
00777
00778     template< typename IType, typename VType >
00779     struct is_operator< operators::argmax< IType, VType > > {
00780         static const constexpr bool value = true;
00781     };
00782
00783     template< typename D1, typename D2, typename D3, enum Backend implementation >
00784     struct is_operator< operators::square_diff< D1, D2, D3, implementation > > {
00785         static const constexpr bool value = true;
00786     };
00787
00788     template< typename IN1, typename IN2, enum Backend implementation >
00789     struct is_operator< operators::zip< IN1, IN2, implementation > > {
00790         static const constexpr bool value = true;
00791     };
00792
00793     template< typename D1, typename D2, typename D3, enum Backend implementation >
00794     struct is_operator< operators::equal_first< D1, D2, D3, implementation > > {
00795         static const constexpr bool value = true;
00796     };
00797
```

```
00798     template< typename D1, typename D2, typename D3 >
00799     struct is_idempotent< operators::min< D1, D2, D3 >, void > {
00800         static const constexpr bool value = true;
00801     };
00802
00803     template< typename D1, typename D2, typename D3 >
00804     struct is_idempotent< operators::max< D1, D2, D3 >, void > {
00805         static const constexpr bool value = true;
00806     };
00807
00808     template< typename D1, typename D2, typename D3 >
00809     struct is_idempotent< operators::any_or< D1, D2, D3 >, void > {
00810         static const constexpr bool value = true;
00811     };
00812
00813     template< typename D1, typename D2, typename D3 >
00814     struct is_idempotent< operators::logical_or< D1, D2, D3 >, void > {
00815         static const constexpr bool value = true;
00816     };
00817
00818     template< typename D1, typename D2, typename D3 >
00819     struct is_idempotent< operators::logical_and< D1, D2, D3 >, void > {
00820         static const constexpr bool value = true;
00821     };
00822
00823     template< typename D1, typename D2, typename D3 >
00824     struct is_idempotent< operators::relu< D1, D2, D3 >, void > {
00825         static const constexpr bool value = true;
00826     };
00827
00828     template< typename D1, typename D2, typename D3 >
00829     struct is_idempotent< operators::left_assign_if< D1, D2, D3 >, void > {
00830         static const constexpr bool value = true;
00831     };
00832
00833     template< typename D1, typename D2, typename D3 >
00834     struct is_idempotent< operators::right_assign_if< D1, D2, D3 >, void > {
00835         static const constexpr bool value = true;
00836     };
00837
00838     template< typename IType, typename VType >
00839     struct is_idempotent< operators::argmin< IType, VType >, void > {
00840         static const constexpr bool value = true;
00841     };
00842
00843     template< typename IType, typename VType >
00844     struct is_idempotent< operators::argmax< IType, VType >, void > {
00845         static const constexpr bool value = true;
00846     };
00847
00848     template< typename OP >
00849     struct is_associative<
00850         OP,
00851         typename std::enable_if< is_operator< OP >::value, void >::type
00852     > {
00853         static constexpr const bool value = OP::is_associative();
00854     };
00855
00856     template< typename OP >
00857     struct is_commutative<
00858         OP,
00859         typename std::enable_if< is_operator< OP >::value, void >::type
00860     > {
00861         static constexpr const bool value = OP::is_commutative();
00862     };
00863
00864     // internal type traits follow
00865
00866     namespace internal {
00867
00868         template< typename D1, typename D2, typename D3, enum Backend implementation >
00869         struct maybe_noop< operators::left_assign_if< D1, D2, D3, implementation > > {
00870             static const constexpr bool value = true;
00871         };
00872
00873         template< typename D1, typename D2, typename D3, enum Backend implementation >
00874         struct maybe_noop< operators::right_assign_if< D1, D2, D3, implementation > > {
00875             static const constexpr bool value = true;
00876         };
00877
00878     } // end namespace grb::internal
00879
00880 } // end namespace grb
00881
00882 #ifdef __DOXYGEN__
00888  #define _DEBUG_NO_IOSTREAM_PAIR_CONVERTER
00889 #endif
```

```
00890
00891 #ifdef _DEBUG
00892  #ifndef _DEBUG_NO_IOSTREAM_PAIR_CONVERTER
00893 namespace std {
00894     template< typename U, typename V >
00895     std::ostream & operator«( std::ostream &out, const std::pair< U, V > &pair ) {
00896         out « "( " « pair.first « ", " « pair.second « " )";
00897         return out;
00898     }
00899 } // end namespace std
00900  #endif
00901 #endif
00902
00903 #endif // end "_H_GRB_OPERATORS"
00904
```

10.75 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.

10.75.1 Detailed Description

Defines the various phases an ALP/GraphBLAS primitive may be executed with.

Author

A. N. Yzelman

10.76 phase.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
```

```
00017
00026 #ifndef _H_GRB_PHASE
00027 #define _H_GRB_PHASE
00028
00029
00030 namespace grb {
00031
00152     enum Phase {
00153
00187         RESIZE,
00188
00225         TRY,
00226
00257         EXECUTE
00258
00259     };
00260
00261 } // namespace grb
00262
00263 #endif // end "_H_GRB_PHASE"
00264
```

10.77 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)

10.77.1 Detailed Description

Defines the ALP error codes.

Author

A. N. Yzelman

Date

9–11 August, 2016

10.78 rc.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_RC
00028 #define _H_GRB_RC
00029
00030 #include <string>
00031
00032
00033 namespace grb {
00034
00047     enum RC {
00048
00054         SUCCESS = 0,
00055
00068         PANIC,
00069
00078         OUTOFMEM,
00079
00090         MISMATCH,
00091
00106         OVERLAP,
00107
00119         OVERFLW,
00120
00129         UNSUPPORTED,
00130
00143         ILLEGAL,
00144
00154         FAILED
00155
00156     };
00157
00159     std::string toString( const RC code );
00160
00161 } // namespace grb
00162
00163 #endif
00164
```

10.79 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.

10.79.1 Detailed Description

Provides an ALP semiring.

Author

A. N. Yzelman

Date

15th of March, 2016

10.80 semiring.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_SEMIRING
00028 #define _H_GRB_SEMIRING
00029
00030 #include <graphblas/identities.hpp>
00031 #include <graphblas/monoid.hpp>
00032 #include <graphblas/ops.hpp>
00033
00034
00035 namespace grb {
00036
00185     template< class _OP1, class _OP2, template< typename > class _ID1, template< typename > class _ID2
    >
00186     class Semiring {
00187
00188         static_assert( std::is_same< typename _OP2::D3, typename _OP1::D1 >::value,
00189             "The multiplicative output type must match the left-hand additive "
00190             "input type" );
00191
00192         static_assert( std::is_same< typename _OP1::D2, typename _OP1::D3 >::value,
00193             "The right-hand input type of the additive operator must match its "
00194             "output type" );
00195
00196         static_assert( grb::is_associative< _OP1 >::value,
00197             "Cannot construct a semiring using a non-associative additive "
00198             "operator" );
00199
00200         static_assert( grb::is_associative< _OP2 >::value,
00201             "Cannot construct a semiring using a non-associative multiplicative "
00202             "operator" );
00203
00204         static_assert( grb::is_commutative< _OP1 >::value,
00205             "Cannot construct a semiring using a non-commutative additive "
00206             "operator" );
00207
00208     public:
00210         typedef typename _OP2::D1 D1;
00211
00213         typedef typename _OP2::D2 D2;
00214
00219         typedef typename _OP2::D3 D3;
00220
00225         typedef typename _OP1::D2 D4;
```

```
00226
00228         typedef _OP1 AdditiveOperator;
00229
00231         typedef _OP2 MultiplicativeOperator;
00232
00234         typedef Monoid< _OP1, _ID1 > AdditiveMonoid;
00235
00237         typedef Monoid< _OP2, _ID2 > MultiplicativeMonoid;
00238
00240         template< typename ZeroType >
00241         using Zero = _ID1< ZeroType >;
00242
00244         template< typename OneType >
00245         using One = _ID2< OneType >;
00246
00247     private:
00248         static constexpr size_t D1_bsz = grb::config::SIMD_BLOCKSIZE< D1 >::value();
00249         static constexpr size_t D2_bsz = grb::config::SIMD_BLOCKSIZE< D2 >::value();
00250         static constexpr size_t D3_bsz = grb::config::SIMD_BLOCKSIZE< D3 >::value();
00251         static constexpr size_t D4_bsz = grb::config::SIMD_BLOCKSIZE< D4 >::value();
00252         static constexpr size_t mul_input_bsz = D1_bsz < D2_bsz ? D1_bsz : D2_bsz;
00253
00255         AdditiveMonoid additiveMonoid;
00256
00258         MultiplicativeMonoid multiplicativeMonoid;
00259
00260     public:
00262         static constexpr size_t blocksize_add = D3_bsz < D4_bsz ? D3_bsz : D4_bsz;
00263
00265         static constexpr size_t blocksize_mul = mul_input_bsz < D3_bsz ? mul_input_bsz : D3_bsz;
00266
00268         static constexpr size_t blocksize = blocksize_mul < blocksize_add ? blocksize_mul :
    blocksize_add;
00269
00282         template< typename D >
00283         constexpr D getZero() const {
00284             return additiveMonoid.template getIdentity< D >();
00285         }
00286
00298         template< typename D >
00299         constexpr D getOne() const {
00300             return multiplicativeMonoid.template getIdentity< D >();
00301         }
00302
00308         AdditiveMonoid getAdditiveMonoid() const {
00309             return additiveMonoid;
00310         }
00311
00317         MultiplicativeMonoid getMultiplicativeMonoid() const {
00318             return multiplicativeMonoid;
00319         }
00320
00326         AdditiveOperator getAdditiveOperator() const {
00327             return additiveMonoid.getOperator();
00328         }
00329
00335         MultiplicativeOperator getMultiplicativeOperator() const {
00336             return multiplicativeMonoid.getOperator();
00337         }
00338     };
00339
00340     // overload for GraphBLAS type traits.
00341     template< class _OP1, class _OP2, template< typename > class _ID1, template< typename > class _ID2
    >
00342     struct is_semiring< Semiring< _OP1, _OP2, _ID1, _ID2 > > {
00344         static const constexpr bool value = true;
00345     };
00346
00347     template< class _OP1, class _OP2, template< typename > class _ID1, template< typename > class _ID2
    >
00348     struct has_immutable_nonzeroes< Semiring< _OP1, _OP2, _ID1, _ID2 > > {
00349         static const constexpr bool value = grb::is_semiring< Semiring< _OP1, _OP2, _ID1, _ID2 >
    >::value &&
00350             std::is_same< _OP1, typename grb::operators::logical_or< typename _OP1::D1, typename
    _OP1::D2, typename _OP1::D3 > >::value;
00351     };
00352
00353 } // namespace grb
00354
00355 #endif
00356
```

10.81 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.

10.81.1 Detailed Description

Specifies the ALP algebraic type traits.

Author

A. N. Yzelman

Date

25th of March, 2019

10.82 type_traits.hpp

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00027 #ifndef _H_GRB_TYPE_TRAITS
00028 #define _H_GRB_TYPE_TRAITS
00029
00030 #include <type_traits>
00031
00032
00033 namespace grb {
00034
00046     template< typename T >
00047     struct is_container {
00048
00054         static const constexpr bool value = false;
00055
00056     };
00057
00065     template< typename T >
00066     struct is_semiring {
00067
00073         static const constexpr bool value = false;
00074
00075     };
00076
00084     template< typename T >
00085     struct is_monoid {
00086
00092         static const constexpr bool value = false;
00093
00094     };
00095
00103     template< typename T >
00104     struct is_operator {
00105
00111         static const constexpr bool value = false;
00112     };
00113
00129     template< typename T >
00130     struct is_object {
00131
00135         static const constexpr bool value = is_container< T >::value ||
00136             is_semiring< T >::value ||
00137             is_monoid< T >::value ||
00138             is_operator< T >::value;
00139     };
00140
00160     template< typename T, typename = void >
00161     struct is_idempotent {
00162
00163         static_assert( is_operator< T >::value || is_monoid< T >::value,
00164             "Template argument to grb::is_idempotent must be an operator or a monoid!" );
00165
00167         static const constexpr bool value = false;
00168
00169     };
00170
00178     template< typename Monoid >
00179     struct is_idempotent<
00180         Monoid,
00181         typename std::enable_if< is_monoid< Monoid >::value, void >::type
00182     > {
00183         static const constexpr bool value =
00184             is_idempotent< typename Monoid::Operator >::value;
00185     };
00186
00201     template< typename T, typename = void >
00202     struct is_associative {
```

```
00203
00204         static_assert( is_operator< T >::value || is_monoid< T >::value,
00205             "Template argument should be an ALP binary operator or monoid." );
00206
00208         static const constexpr bool value = false;
00209
00210     };
00211
00219     template< typename Monoid >
00220     struct is_associative<
00221         Monoid,
00222         typename std::enable_if< is_monoid< Monoid >::value, void >::type
00223     > {
00224         static_assert( is_associative< typename Monoid::Operator >::value,
00225             "Malformed ALP monoid encountered" );
00226         static const constexpr bool value = true;
00227     };
00228
00239     template< typename T, typename = void >
00240     struct is_commutative {
00241
00242         static_assert( is_operator< T >::value || is_monoid< T >::value,
00243             "Template argument should be an ALP binary operator or monoid." );
00244
00246         static const constexpr bool value = false;
00247
00248     };
00249
00257     template< typename Monoid >
00258     struct is_commutative<
00259         Monoid,
00260         typename std::enable_if< is_monoid< Monoid >::value, void >::type
00261     > {
00263         static const constexpr bool value =
00264             is_commutative< typename Monoid::Operator >::value;
00265     };
00266
00278     template< typename T >
00279     struct has_immutable_nonzeroes {
00280
00281         static_assert( is_semiring< T >::value,
00282             "Template argument to grb::has_immutable_nonzeroes must be a "
00283             "semiring!" );
00284
00286         static const constexpr bool value = false;
00287
00288     };
00289
00290     namespace internal {
00291
00304         template< typename OP >
00305         struct maybe_noop {
00306             static_assert( is_operator< OP >::value,
00307                 "Argument to internal::maybe_noop must be an operator."
00308             );
00309             static const constexpr bool value = false;
00310         };
00311
00312     } // end namespace grb::internal
00313
00314 } // namespace grb
00315
00316 #endif // end _H_GRB_TYPE_TRAITS
00317
```

10.83 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.

10.83.1 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.

10.83.2 Typedef Documentation

10.83.2.1 blas_sparse_matrix

```
typedef void* blas_sparse_matrix
```

A sparse matrix.

See the SparseBLAS paper for the full specification.

10.83.3 Enumeration Type Documentation

10.83.3.1 blas_order_type

```
enum blas_order_type
```

The supported dense storages.

See the SparseBLAS paper for the full specification.

10.83.3.2 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`.

10.83.4 Function Documentation

10.83.4.1 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.

10.83.4.2 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.

10.83.4.3 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.

10.83.4.4 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.

10.83.4.5 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.

10.83.4.6 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.

10.83.4.7 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.

10.83.4.8 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.

10.83.4.9 BLAS_usds()

```
int BLAS_usds (
              blas_sparse_matrix A )
```

Frees a given matrix.

See the SparseBLAS paper for the full specification.

10.83.4.10 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.

10.83.4.11 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.

10.83.4.12 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.

10.83.4.13 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.

10.83.4.14 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.

10.83.4.15 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

- $C \rightarrow \alpha AB + C$,
- $C \rightarrow \alpha A^T B + C$,
- $C \rightarrow \alpha AB^T + C$, or
- $C \rightarrow \alpha A^T B^T + C$.

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.
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.

10.83.4.16 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.
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.

10.83.4.17 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.

10.84 blas_sparse.h

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00026 #ifndef _H_ALP_SPARSEBLAS_NIST
00027 #define _H_ALP_SPARSEBLAS_NIST
00028
00029 #include "blas_sparse_vec.h"
00030
00031 #ifdef __cplusplus
00032 extern "C" {
00033 #endif
00034
00042 enum blas_trans_type {
00043     blas_no_trans = 0,
00044     blas_trans,
```

```
00045     blas_conj_trans
00046 };
00047
00053 enum blas_order_type {
00054     blas_rowmajor,
00055     blas_colmajor
00056 };
00057
00067 typedef void * blas_sparse_matrix;
00068
00077 blas_sparse_matrix BLAS_duscr_begin( const int m, const int n );
00078
00084 int BLAS_duscr_insert_entry(
00085     blas_sparse_matrix A,
00086     const double val,
00087     const int row, const int col
00088 );
00089
00095 int BLAS_duscr_insert_entries(
00096     blas_sparse_matrix A,
00097     const int nnz,
00098     const double * vals, const int * rows, const int * cols
00099 );
00100
00106 int BLAS_duscr_insert_col(
00107     blas_sparse_matrix A,
00108     const int j, const int nnz,
00109     const double * vals, const int * rows
00110 );
00111
00117 int BLAS_duscr_insert_row(
00118     blas_sparse_matrix A,
00119     const int i, const int nnz,
00120     const double * vals, const int * cols
00121 );
00122
00129 int BLAS_duscr_end( blas_sparse_matrix A );
00130
00136 int BLAS_usds( blas_sparse_matrix A );
00137
00147 int BLAS_dusmv(
00148     const enum blas_trans_type transa,
00149     const double alpha, const blas_sparse_matrix A,
00150     const double * const x, int incx,
00151     double * const y, const int incy
00152 );
00153
00163 int BLAS_dusmm(
00164     const enum blas_order_type order,
00165     const enum blas_trans_type transa,
00166     const int nrhs,
00167     const double alpha, const blas_sparse_matrix A,
00168     const double * B, const int ldb,
00169     const double * C, const int ldc
00170 );
00171
00193 int EXTBLAS_dusmsv(
00194     const enum blas_trans_type transa,
00195     const double alpha, const blas_sparse_matrix A,
00196     const extblas_sparse_vector x,
00197     extblas_sparse_vector y
00198 );
00199
00223 int EXTBLAS_dusmsm(
00224     const enum blas_trans_type transa,
00225     const double alpha, const blas_sparse_matrix A,
00226     const enum blas_trans_type transb, const blas_sparse_matrix B,
00227     blas_sparse_matrix C
00228 );
00229
00242 int EXTBLAS_dusm_nz( const blas_sparse_matrix A, int * nz );
00243
00259 int EXTBLAS_dusm_open( const blas_sparse_matrix A );
00260
00290 int EXTBLAS_dusm_get(
00291     const blas_sparse_matrix A,
00292     double * value, int * row, int * col
00293 );
00294
00306 int EXTBLAS_dusm_close( const blas_sparse_matrix A );
00307
00319 int EXTBLAS_dusm_clear( blas_sparse_matrix A );
00320
00330 int EXTBLAS_free();
00331
00332 #ifdef __cplusplus
00333 } // end extern "C"
```

```
00334 #endif
00335
00336 #endif // end _H_ALP_SPARSEBLAS_NIST
00337
```

10.85 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.

10.85.1 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.

10.85.2 Typedef Documentation

10.85.2.1 extblas_sparse_vector

```
typedef void* extblas_sparse_vector
```

A sparse vector.

This is an implementation-specific extension.

10.85.3 Function Documentation

10.85.3.1 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.

10.85.3.2 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.

10.85.3.3 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.

10.85.3.4 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.

10.85.3.5 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.

10.85.3.6 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.

10.85.3.7 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.

10.85.3.8 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.

10.85.3.9 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.

10.86 blas_sparse_vec.h

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00026 #ifndef _H_ALP_SPARSEBLAS_EXT_VEC
00027 #define _H_ALP_SPARSEBLAS_EXT_VEC
00028
00029 #ifdef __cplusplus
00030 extern "C" {
00031 #endif
00032
00034 typedef void * extblas_sparse_vector;
00035
00045 extblas_sparse_vector EXTBLAS_dusv_begin( const int n );
00046
00063 int EXTBLAS_dusv_insert_entry(
00064     extblas_sparse_vector x,
00065     const double val,
00066     const int index
00067 );
00068
00081 int EXTBLAS_dusv_end( extblas_sparse_vector x );
00082
00095 int EXTBLAS_dusvds( extblas_sparse_vector x );
00096
00110 int EXTBLAS_dusv_nz( const extblas_sparse_vector x, int * nz );
00111
00127 int EXTBLAS_dusv_open( const extblas_sparse_vector x );
00128
00157 int EXTBLAS_dusv_get(
00158     const extblas_sparse_vector x,
00159     double * const val, int * const ind
00160 );
00161
00173 int EXTBLAS_dusv_close( const extblas_sparse_vector x );
00174
00186 int EXTBLAS_dusv_clear( extblas_sparse_vector x );
00187
00188 #ifdef __cplusplus
00189 } // end extern "C"
00190 #endif
00191
00192 #endif // end '_H_ALP_SPARSEBLAS_EXT_VEC'
00193
```

10.87 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).*

10.87.1 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.

10.87.2 Function Documentation

10.87.2.1 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. $y \to y + \alpha A x$, or
2. $y \to y + \alpha A^T x$.

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.
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.
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.

10.87.2.2 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.

10.87.2.3 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.

Parameters

in	*alpha*	Pointer to the scalar α.
in	*matdescra*	Has several entries. Going from first to last: Either 'G', 'S', 'H', 'T', 'A', or 'D' (similar to MatrixMarket) 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.
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.
in	*ldc*	Leading dimension of *c*. If in row-major format, this should be n. If in column-major format, this should be m.

10.87.2.4 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.
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 and that the contents of the row offset arrays have not been modified. It also assumes that the column
		index and value arrays are (now) of sufficient size to hold the output.

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*.
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*.
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.

10.88 spblas.h

Go to the documentation of this file.

```
00001
00002 /*
00003  *   Copyright 2021 Huawei Technologies Co., Ltd.
00004  *
00005  * Licensed under the Apache License, Version 2.0 (the "License");
00006  * you may not use this file except in compliance with the License.
00007  * You may obtain a copy of the License at
00008  *
00009  *     http://www.apache.org/licenses/LICENSE-2.0
00010  *
00011  * Unless required by applicable law or agreed to in writing, software
00012  * distributed under the License is distributed on an "AS IS" BASIS,
00013  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00014  * See the License for the specific language governing permissions and
00015  * limitations under the License.
00016  */
00017
00034 #ifndef _H_ALP_SPBLAS
00035 #define _H_ALP_SPBLAS
00036
00037 #include "blas_sparse_vec.h"
00038
00039 #ifdef __cplusplus
00040 extern "C" {
00041 #endif
00042
00064 void spblas_dcsrgemv(
00065     const char * transa,
00066     const int * m,
00067     const double * a, const int * ia, const int * ja,
00068     const double * x,
00069     double * y
00070 );
00071
00101 void spblas_dcsrmm(
00102     const char * transa,
00103     const int * m, const int * n, const int * k,
00104     const double * alpha,
00105     const char * matdescra, const double * val, const int * indx,
00106     const int * pntrb, const int * pntre,
00107     const double * b, const int * ldb,
00108     const double * beta,
00109     double * c, const int * ldc
00110 );
00111
00152 void spblas_dcsrmultcsr(
00153     const char * trans, const int * request, const int * sort,
00154     const int * m, const int * n, const int * k,
00155     double * a, int * ja, int * ia,
00156     double * b, int * jb, int * ib,
```

```
00157     double * c, int * jc, int * ic,
00158     const int * nzmax, int * info
00159 );
00160
00192 void extspblas_dcsrmultsv(
00193     const char * trans, const int * request,
00194     const int * m, const int * n,
00195     const double * a, const int * ja, const int * ia,
00196     const extblas_sparse_vector x,
00197     extblas_sparse_vector y
00198 );
00199
00204 void extspblas_free();
00205
00206 #ifdef __cplusplus
00207 } // end extern "C"
00208 #endif
00209
00210 #endif // end _H_ALP_SPBLAS
00211
```

Index