Algebraic Programming

Portable, high-performance, and high-productivity programming

Albert-Jan N. Yzelman

Computing Systems Laboratory
Zürich Research Center, Switzerland

NUMA, dept. CW, KU Leuven, 7th of September, 2023

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;

- **Humble** programmers: achieve maximum productivity.

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;
  - domain, lower bounds, algorithms, coding, *and* hardware experts

- **Humble** programmers: achieve maximum productivity.

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;
  - domain, lower bounds, algorithms, coding, *and* hardware experts
  - increasingly complex: many-core, heterogeinity, **deeper NUMA** effects, memory walls, and **low memory capacity** per core.

- **Humble** programmers: achieve maximum productivity.

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;
  - domain, lower bounds, algorithms, coding, *and* hardware experts
  - increasingly complex: many-core, heterogeinity, **deeper NUMA** effects, memory walls, and **low memory capacity** per core.

- **Humble** programmers: achieve maximum productivity.
  - easy-to-use, "*scalable*" programming: MapReduce, Spark, ...

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;
  - domain, lower bounds, algorithms, coding, *and* hardware experts
  - increasingly complex: many-core, heterogeinity, **deeper NUMA** effects, memory walls, and **low memory capacity** per core.

- **Humble** programmers: achieve maximum productivity.
  - easy-to-use, "*scalable*" programming: MapReduce, Spark, ...
  - 100% of peak performance not absolutely required

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;
  - domain, lower bounds, algorithms, coding, *and* hardware experts
  - increasingly complex: many-core, heterogeinity, **deeper NUMA** effects, memory walls, and **low memory capacity** per core.

- **Humble** programmers: achieve maximum productivity.
  - easy-to-use, "*scalable*" programming: MapReduce, Spark, ...
  - 100% of peak performance not absolutely required
  - typically **sequential, data-centric, reliable, & automatic**

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;

  - domain, lower bounds, algorithms, coding, *and* hardware experts
  - increasingly complex: many-core, heterogeinity, **deeper NUMA** effects, memory walls, and **low memory capacity** per core.

- **Humble** programmers: achieve maximum productivity.

  - easy-to-use, "*scalable*" programming: MapReduce, Spark, ...
  - 100% of peak performance not absolutely required
  - typically **sequential, data-centric, reliable, & automatic**

Increasingly many hardware targets,

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;

  - domain, lower bounds, algorithms, coding, *and* hardware experts
  - increasingly complex: many-core, heterogeinity, **deeper NUMA** effects, memory walls, and **low memory capacity** per core.

- **Humble** programmers: achieve maximum productivity.

  - easy-to-use, "*scalable*" programming: MapReduce, Spark, ...
  - 100% of peak performance not absolutely required
  - typically **sequential, data-centric, reliable, & automatic**

Increasingly many hardware targets,

increasingly heterogeneous hardware:

Humble and Hero Programming

- **Hero** programmers: achieve maximum efficiency;
  - domain, lower bounds, algorithms, coding, *and* hardware experts
  - increasingly complex: many-core, heterogeinity, **deeper NUMA** effects, memory walls, and **low memory capacity** per core.

- **Humble** programmers: achieve maximum productivity.
  - easy-to-use, "*scalable*" programming: MapReduce, Spark, ...
  - 100% of peak performance not absolutely required
  - typically **sequential, data-centric, reliable, & automatic**

Increasingly many hardware targets,

increasingly heterogeneous hardware:

a software productivity crisis is looming

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?
- standard interfaces may not map well to all architectures.

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?
- standard interfaces may not map well to all architectures.

Compile humble code to new architectures:

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?
- standard interfaces may not map well to all architectures.

Compile humble code to new architectures:

- compiler would need to 'think' about algorithms;

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?
- standard interfaces may not map well to all architectures.

Compile humble code to new architectures:

- compiler would need to 'think' about algorithms;
- best algorithms may depend on run-time conditions;

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?
- standard interfaces may not map well to all architectures.

Compile humble code to new architectures:

- compiler would need to ‘think’ about algorithms;
- best algorithms may depend on run-time conditions;
- user-driven compilation and DSLs: less humble, how many DSLs?

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?
- standard interfaces may not map well to all architectures.

Compile humble code to new architectures:

- compiler would need to 'think' about algorithms;
- best algorithms may depend on run-time conditions;
- user-driven compilation and DSLs: less humble, how many DSLs?

Rewrite applications using new framework(s) and/or DSL(s):

- rewrite software or use older/slower hardware?

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?
- standard interfaces may not map well to all architectures.

Compile humble code to new architectures:

- compiler would need to 'think' about algorithms;
- best algorithms may depend on run-time conditions;
- user-driven compilation and DSLs: less humble, how many DSLs?

Rewrite applications using new framework(s) and/or DSL(s):

- rewrite software or use older/slower hardware?

Many existing software, many workload domains, many architectures:

Compilers, libraries, and applications

New hardware, provide **libraries** with standard APIs (e.g., BLAS):

- if many hardware targets, how many heroes do we need?
- standard interfaces may not map well to all architectures.

Compile humble code to new architectures:

- compiler would need to 'think' about algorithms;
- best algorithms may depend on run-time conditions;
- user-driven compilation and DSLs: less humble, how many DSLs?

Rewrite applications using new framework(s) and/or DSL(s):

- rewrite software or use older/slower hardware?

Many existing software, many workload domains, many architectures:

- **re-define compiler, library, and application boundaries**!

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Principles:

- explicitly annotate computations with **algebraic information**;

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Principles:

- explicitly annotate computations with **algebraic information**;
- allow **compile-time introspection** of algebraic information;

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Principles:

- explicitly annotate computations with **algebraic information**;
- allow **compile-time introspection** of algebraic information;
- automatically **optimise** code based on algebraic information;

Algebraic Programming

Algebraic Programming, or **ALP** for short:

- sequential, data-centric, standard C++;
- similar to the Standard Template Library (STL), thus
- **humble**, yet also **auto-parallelising** and **high performance**.

Principles:

- explicitly annotate computations with **algebraic information**;
- allow **compile-time introspection** of algebraic information;
- automatically **optimise** code based on algebraic information;
- allow only **scalable** expressions.

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;
2) structures:
3) primitives:

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;
2) structures: binary operators, monoids, and semirings; and
3) primitives:

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1. containers: scalars, vectors, and matrices;
2. structures: binary operators, monoids, and semirings; and
3. primitives: eWiseApply, reduction into scalar or vector (fold), mxv.

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;
2) structures: binary operators, monoids, and semirings; and
3) primitives: eWiseApply, reduction into scalar or vector (fold), mxv.

Containers are similar to the standard template library (STL):

```
grb::Vector< double > x( n ), y( n ), z( n );
grb::Matrix< double > A( n, n );
```

Basics

Three ALP concepts: algebraic *containers*, *structures*, and *primitives*.

Example: **sparse linear algebra**, ALP/GraphBLAS

1) containers: scalars, vectors, and matrices;
2) structures: binary operators, monoids, and semirings; and
3) primitives: eWiseApply, reduction into scalar or vector (fold), mxv.

Containers are similar to the standard template library (STL):

```
grb::Vector< double > x( n ), y( n ), z( n );
grb::Matrix< double > A( n, n );
```

Elements may be **any POD type**, containers have **capacities** and **IDs**:

```
grb::Vector< std::pair< int, double > > pairs( n );
grb::Vector< bool > s( n, 1 );        // nz cap: one
grb::Matrix< void > L( n, n, nz ); // nz cap: nz
std::cout << "s has ID " << grb::getID( s ) << "\n";
```

Basics

Algebraic structures are types. E.g., $\min: D_1 \times D_2 \to D_3$ reads

```
grb::operators::min< double, int, double > minOp;
```

Basics

Algebraic structures are types. E.g., min : $D_1 \times D_2 \to D_3$ reads

```
grb::operators::min< double, int, double > minOp;
```

More complex structures may be **composed**:

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

Basics

Algebraic structures are types. E.g., min : $D_1 \times D_2 \to D_3$ reads

```
grb::operators::min< double, int, double > minOp;
```

More complex structures may be **composed**:

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

grb::Semiring<
    grb::operators::add< double >,
    grb::operators::mul< double >,
    grb::identities::zero,
    grb::identities::one
> mySemiring;
```

Basics

Algebraic primitives operate on algebraic containers:

- grb :: set(x, 1.0); $\quad // \; x_i = 1, \; \forall i$
- grb :: setElement(y, 3.0, n/2); $\quad // \; y_{n/2} = 3$

Basics

Algebraic primitives operate on algebraic containers:

- grb :: set(x, 1.0); // $x_i = 1, \ \forall i$
- grb :: setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may change based on **required** algebraic structures:

- grb :: eWiseApply(z, x, y, minOp); // $z_i = \min\{x_i, y_i\}$, for $i = n/2$
- grb :: eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, y_i\}, \ \forall i$

Basics

Algebraic primitives operate on algebraic containers:

- grb :: set(x, 1.0); // $x_i = 1, \ \forall i$
- grb :: setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may change based on **required** algebraic structures:

- grb :: eWiseApply(z, x, y, minOp); // $z_i = \min\{x_i, y_i\}$, for $i = n/2$
- grb :: eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, y_i\}, \ \forall i$
- grb :: eWiseApply(z, x, y, addMon);// $z_i = x_i + y_i, \ \forall i$

Basics

Algebraic primitives operate on algebraic containers:

- grb::set(x, 1.0); // $x_i = 1, \ \forall i$
- grb::setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may change based on **required** algebraic structures:

- grb::eWiseApply(z, x, y, minOp); // $z_i = \min\{x_i, y_i\}$, for $i = n/2$
- grb::eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, y_i\}, \ \forall i$
- grb::eWiseApply(z, x, y, addMon);// $z_i = x_i + y_i, \ \forall i$
- grb::mxv(y, A, x, mySemiring); // y += Ax; in-place

Basics

Algebraic primitives operate on algebraic containers:

- grb :: set(x, 1.0); // $x_i = 1, \forall i$
- grb :: setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may change based on **required** algebraic structures:

- grb :: eWiseApply(z, x, y, minOp); // $z_i = \min\{x_i, y_i\}$, for $i = n/2$
- grb :: eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, y_i\}, \ \forall i$
- grb :: eWiseApply(z, x, y, addMon);// $z_i = x_i + y_i, \ \forall i$
- grb :: mxv(y, A, x, mySemiring); // y += Ax; in-place

All primitives except '**getters**' such as grb :: { size , nrows, nnz, capacity }:

Basics

Algebraic primitives operate on algebraic containers:

- grb :: set(x, 1.0); // $x_i = 1, \ \forall i$
- grb :: setElement(y, 3.0, n/2); // $y_{n/2} = 3$

Semantics may change based on **required** algebraic structures:

- grb :: eWiseApply(z, x, y, minOp); // $z_i = \min\{x_i, y_i\}$, for $i = n/2$
- grb :: eWiseApply(z, x, x, minOp); // $z_i = \min\{x_i, y_i\}, \ \forall i$
- grb :: eWiseApply(z, x, y, addMon);// $z_i = x_i + y_i, \ \forall i$
- grb :: mxv(y, A, x, mySemiring); // y += Ax; in-place

All primitives except '**getters**' such as grb :: { size , nrows, nnz, capacity }:

- allow output **masks**, **descriptors**, and **phase** arguments;
- return **error codes** such as for mismatching dimensions.

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

- grb :: is_associative< Operator >::value, `true` iff $(a \odot b) \odot c = a \odot (b \odot c)$;
- grb :: is_idempotent< Operator >::value, `true` iff $a \odot a = a$;
- grb :: is_monoid< T >::value, `true` iff T is a monoid;
- ...

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

- grb :: is_associative < Operator >::value, `true` iff $(a \odot b) \odot c = a \odot (b \odot c)$;
- grb :: is_idempotent< Operator >::value, `true` iff $a \odot a = a$;
- grb :: is_monoid< T >::value, `true` iff T is a monoid;
- ...

Algebraic type traits transfer to richer algebraic structures:

- grb :: is_commutative< grb::operators::add< **double** > >::value? ✓

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

- grb :: is_associative < Operator >::value, `true` iff $(a \odot b) \odot c = a \odot (b \odot c)$;
- grb :: is_idempotent< Operator >::value, `true` iff $a \odot a = a$;
- grb :: is_monoid< T >::value, `true` iff T is a monoid;
- ...

Algebraic type traits transfer to richer algebraic structures:

- grb :: is_commutative< grb::operators::add< **double** > >::value? ✓,

  therefore

  ```
  is_commutative< Monoid<
     operators::add< double >, identities::zero >
  >::value?
  ```
  ✓

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

- grb :: is_associative < Operator >::value, `true` iff $(a \odot b) \odot c = a \odot (b \odot c)$;
- grb :: is_idempotent< Operator >::value, `true` iff $a \odot a = a$;
- grb :: is_monoid< T >::value, `true` iff T is a monoid;
- ...

Algebraic type traits transfer to richer algebraic structures:

- grb :: is_commutative< grb::operators::add< **double** > >::value? ✓, therefore

  is_commutative< Monoid<
  operators::add< **double** >, identities::zero >
  >::value? ✓

These are all **compile-time constant** (through C++11 constexpr):

- similar to the standard C++11 *type traits*.

Algebraic type traits

Algebraic type traits help

- detect programmer errors,
- decide which optimisations are applicable, and
- reject expressions without recipe for auto-parallelisation.

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities ::one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities :: one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators :: right_assign<**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities :: one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators :: right_assign <**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗
- reducing a sparse vector to a scalar without a monoid structure.
  - operators :: add< **double** > addOp;
  - **double** alpha = 0; foldl (alpha , x, addOp);
  - is_monoid< addOp >::value? ✗

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities :: one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators :: right_assign <**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗
- reducing a sparse vector to a scalar without a monoid structure.
  - operators :: add< **double** > addOp;
  - **double** alpha = 0; foldl (alpha , x, addOp);
  - is_monoid< addOp >::value? ✗, since
    parallelisation requires identity *and* associativity!
    foldl (alpha , x, addMon); ✓

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - Monoid< operators::divide<**int**>, identities :: one > myMonoid;
  - is_associative < operators :: divide < **int** > >::value? ✗
- composing a semiring using a non-commutative additive monoid;
  - Semiring< operators :: right_assign <**double**>, ... > mySemiring;
  - is_commutative< operators::right_assign<**double**> >::value? ✗
- reducing a sparse vector to a scalar without a monoid structure.
  - operators :: add< **double** > addOp;
  - **double** alpha = 0; foldl (alpha , x, addOp);
  - is_monoid< addOp >::value? ✗, since
    parallelisation requires identity *and* associativity!

      foldl (alpha , x, addMon); ✓

Above errors are all **compile-time** (through C++11 static_assert), with

Algebraic type traits

For example, **algebraic type traits prevent**

- creating a monoid from non-associative operator;
  - `Monoid< operators::divide<int>, identities::one > myMonoid;`
  - `is_associative< operators::divide< int > >::value?` ✗
- composing a semiring using a non-commutative additive monoid;
  - `Semiring< operators::right_assign<double>, ... > mySemiring;`
  - `is_commutative< operators::right_assign<double> >::value?` ✗
- reducing a sparse vector to a scalar without a monoid structure.
  - `operators::add< double > addOp;`
  - `double alpha = 0; foldl( alpha, x, addOp );`
  - `is_monoid< addOp >::value?` ✗, since
    parallelisation requires identity *and* associativity!

    `foldl( alpha, x, addMon );` ✓

Above errors are all **compile-time** (through C++11 `static_assert`), with

clear error messages.

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$?

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$? Replace eWiseApply with grb::set(z, x);!

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb :: eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$? Replace eWiseApply with grb :: set(z, x);!
  - every time S is **idempotent**;

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb :: eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$? Replace eWiseApply with grb :: set(z, x);!
  - every time S is **idempotent**;
- ...

Algebraic type traits

E.g. (ct'd), algebraic type traits allow, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot =$ min? Replace eWiseApply with grb::set(z, x);!
  - every time S is **idempotent**;
- ...

These optimisations are applied at compile-time,

Algebraic type traits

E.g. (ct'd), **algebraic type traits allow**, for algebraic structure S, to

- split up and parallelise reduce operations
  - if S is **associative** and has an **identity**;
- reorder computation to improve cache hit rates
  - if S is **commutative**;
- replace primitives with cheaper ones:
  - grb::eWiseApply(z, x, x, S); sets $z_i = x_i \odot x_i$;
  - $\odot = \min$? Replace eWiseApply with grb::set(z, x);!
  - every time S is **idempotent**;
- ...

These optimisations are applied at compile-time,

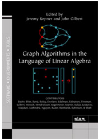
without requiring programmer knowledge or intervention.

Ex.: Y & Bisseling '10; Y & Roose '14; Y, Bisseling, Roose, Meerbergen '14; Y, Roose, Meerbergen '14; ...

Historical context

- The Design and Analysis of Computer Algorithms, Aho, Hopcroft, Ullman (1974)
- Introduction to Algorithms (first edition only), Cormen, Leiserson, Rivest (1990)
- **Elements of Programming**, Alexander Stepanov & Paul McJones (2009)
- Graph Algorithms in the Language of Linear Algebra, Jeremy Kepner & John Gilbert (2011)
- From Mathematics to Generic Programming, Alexander Stepanov & Daniel Rose (2015)
- **GraphBLAS.org**, following work by Kepner & Gilbert, Kepner, Gilbert, Buluç, Mattson, et alii (2016)
- A C++ GraphBLAS, Y. et al. (2017–2020)
- **Algebraic Programming** (2021 onwards)
- ...

GraphBLAS

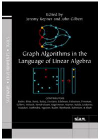


`GraphBLAS.org`; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

- for example, $y = A^k x$, parametrised in a semiring:

```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
              grb::Vector< VectorT >   &y,
        const grb::Matrix< NonzeroT > &A, const size_t k,
        const grb::Vector< VectorT >   &x,
              grb::Vector< VectorT >   &buffer,
        const Semiring                 &ring = Semiring()
) {
        // error checking and error propagation omitted
        grb::vxm( y, x, A, ring );
        for( size_t i = 1; i < k; ++i ) {
                std::swap( y, buffer );
                grb::vxm( y, buffer, A, ring );
        }
}
```

GraphBLAS



GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

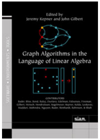
- for example, $y = A^k x$, parametrised in a semiring:

```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
              grb::Vector< VectorT >  &y,
        const grb::Matrix< NonzeroT > &A, const size_t k,
        const grb::Vector< VectorT >  &x,
              grb::Vector< VectorT >  &buffer,
        const Semiring                &ring = Semiring()
) {
        // error checking and error propagation omitted
        grb::vxm( y, x, A, ring );
        for( size_t i = 1; i < k; ++i ) {
                std::swap( y, buffer );
                grb::vxm( y, buffer, A, ring );
        }
}
```

Solve different problems with different semirings:

- plus-times: numerical linear algebra,

GraphBLAS



GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

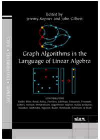
- for example, $y = A^k x$, parametrised in a semiring:

```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
              grb::Vector< VectorT >  &y,
        const grb::Matrix< NonzeroT > &A, const size_t k,
        const grb::Vector< VectorT >  &x,
              grb::Vector< VectorT >  &buffer,
        const Semiring                &ring = Semiring()
) {
        // error checking and error propagation omitted
        grb::vxm( y, x, A, ring );
        for( size_t i = 1; i < k; ++i ) {
                std::swap( y, buffer );
                grb::vxm( y, buffer, A, ring );
        }
}
```

Solve different problems with different semirings:

- plus-times: numerical linear algebra,
- Boolean: reachability / connectivity,

GraphBLAS



GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

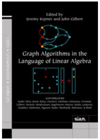
- for example, $y = A^k x$, parametrised in a semiring:

```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
              grb::Vector< VectorT >  &y,
        const grb::Matrix< NonzeroT > &A, const size_t k,
        const grb::Vector< VectorT >  &x,
              grb::Vector< VectorT >  &buffer,
        const Semiring                &ring = Semiring()
) {
        // error checking and error propagation omitted
        grb::vxm( y, x, A, ring );
        for( size_t i = 1; i < k; ++i ) {
                std::swap( y, buffer );
                grb::vxm( y, buffer, A, ring );
        }
}
```

Solve different problems with different semirings:

- plus-times: numerical linear algebra,
- Boolean: reachability / connectivity,
- min-plus: shortest paths,

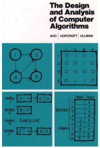
GraphBLAS



GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

- for example, $y = A^k x$, parametrised in a semiring:

```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
                grb::Vector< VectorT >  &y,
        const grb::Matrix< NonzeroT > &A, const size_t k,
        const grb::Vector< VectorT >  &x,
              grb::Vector< VectorT >  &buffer,
        const Semiring                &ring = Semiring()
) {
        // error checking and error propagation omitted
        grb::vxm( y, x, A, ring );
        for( size_t i = 1; i < k; ++i ) {
                std::swap( y, buffer );
                grb::vxm( y, buffer, A, ring );
        }
}
```



Solve different problems with different semirings:

- plus-times: numerical linear algebra,
- Boolean: reachability / connectivity,
- min-plus: shortest paths,
- ...and more – see e.g. Aho, Hopcroft, and Ullman '74; Kepner & Gilbert '11.

Example

Single-source shortest-paths (SSSP) using a min-plus semiring:

- graph represented by its $n \times n$ adjacency matrix A

Example

Single-source shortest-paths (SSSP) using a min-plus semiring:

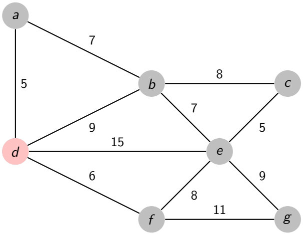
- graph represented by its $n \times n$ adjacency matrix A
- SSSP via $A^k x$ using the semiring $(N_0, \min, +, \infty, 0)$:

Example

Single-source shortest-paths (SSSP) using a min-plus semiring:

- graph represented by its $n \times n$ adjacency matrix A
- SSSP via $A^k x$ using the semiring $(N_0, \min, +, \infty, 0)$:

$$\underbrace{\begin{pmatrix} \\ \\ \\ 0 \\ \\ \\ \\ \end{pmatrix}}_{x+=Ax} += \underbrace{\begin{pmatrix} x & 7 & & 5 & & & \\ 7 & x & 8 & 9 & 7 & & \\ & 8 & x & & 5 & & \\ 5 & 9 & & x & 15 & 6 & \\ & 7 & 5 & 15 & x & 8 & 9 \\ & & & 6 & 8 & x & 11 \\ & & & & 9 & 11 & x \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \\ \\ \\ 0 \\ \\ \\ \\ \end{pmatrix}}_{x}$$



Ax: shortest distances within *one* hop:

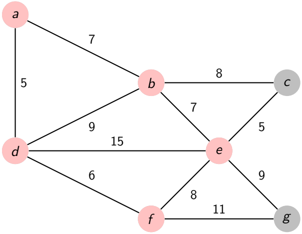
- x contains one nonzero;

Example

Single-source shortest-paths (SSSP) using a min-plus semiring:

- graph represented by its $n \times n$ adjacency matrix A
- SSSP via $A^k x$ using the semiring $(N_0, \min, +, \infty, 0)$:

$$\underbrace{\begin{pmatrix} \\ \\ \\ 0 \\ \\ \\ \\ \end{pmatrix}}_{x+=Ax} += \underbrace{\begin{pmatrix} x & 7 & & \mathbf{5} & & & \\ 7 & x & 8 & \mathbf{9} & 7 & & \\ & 8 & x & & 5 & & \\ 5 & 9 & & x & 15 & 6 & \\ & 7 & 5 & \mathbf{15} & x & 8 & 9 \\ & & & \mathbf{6} & 8 & x & 11 \\ & & & & 9 & 11 & x \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \\ \\ \\ 0 \\ \\ \\ \\ \end{pmatrix}}_{x}$$



Ax: shortest distances within *one* hop:

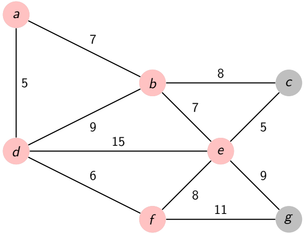
- x contains one nonzero, select corresponding adjacent vertices;

Example

Single-source shortest-paths (SSSP) using a min-plus semiring:

- graph represented by its $n \times n$ adjacency matrix A
- SSSP via $A^k x$ using the semiring $(N_0, \min, +, \infty, 0)$:

$$\underbrace{\begin{pmatrix} 5 \\ 9 \\ \\ 0 \\ 15 \\ 6 \\ \\ \end{pmatrix}}_{x+=Ax} += \underbrace{\begin{pmatrix} x & 7 & & \mathbf{5} & & & \\ 7 & x & 8 & \mathbf{9} & 7 & & \\ & 8 & x & & 5 & & \\ 5 & 9 & & x & 15 & 6 & \\ & 7 & 5 & \mathbf{15} & x & 8 & 9 \\ & & & \mathbf{6} & 8 & x & 11 \\ & & & & 9 & 11 & x \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} \\ \\ \\ 0 \\ \\ \\ \\ \end{pmatrix}}_{x}$$



Ax: shortest distances within *one* hop:

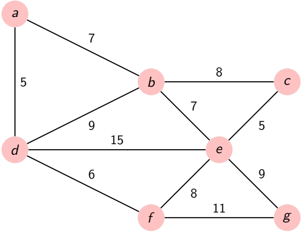
- x contains one nonzero, select corresponding adjacent vertices;
- compute sum with inputs, reduce into output using min.

Example

Single-source shortest-paths (SSSP) using a min-plus semiring:

- graph represented by its $n \times n$ adjacency matrix A
- SSSP via $A^k x$ using the semiring $(N_0, \min, +, \infty, 0)$:

$$\underbrace{\begin{pmatrix} \\ \\ \\ {\color{blue}0} \\ \\ \\ \\ \end{pmatrix}}_{x += A(Ax)} += \underbrace{\begin{pmatrix} \mathrm{x} & 7 & & 5 & & & \\ 7 & \mathrm{x} & 8 & 9 & 7 & & \\ & 8 & \mathrm{x} & & 5 & & \\ 5 & 9 & & \mathrm{x} & 15 & 6 & \\ & 7 & 5 & 15 & \mathrm{x} & 8 & 9 \\ & & & 6 & 8 & \mathrm{x} & 11 \\ & & & & 9 & 11 & \mathrm{x} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} 5 \\ 9 \\ \\ 0 \\ 15 \\ 6 \\ \\ \end{pmatrix}}_{Ax}$$



A^2x: shortest distances within *two* hops:

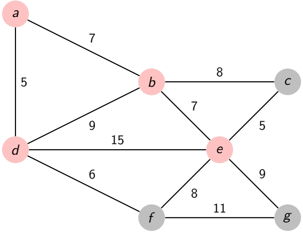
- **swap** input and output vectors, repeat procedure;

Example

Single-source shortest-paths (SSSP) using a min-plus semiring:

- graph represented by its $n \times n$ adjacency matrix A
- SSSP via $A^k x$ using the semiring $(N_0, \min, +, \infty, 0)$:

$$\underbrace{\begin{pmatrix} 5 \\ \mathbf{9} \\ 17 \\ \color{blue}{0} \\ 14 \\ 6 \\ 17 \end{pmatrix}}_{x += A(Ax)} += \underbrace{\begin{pmatrix} \mathrm{x} & 7 & & 5 & & & \\ \mathbf{7} & \mathrm{x} & \mathbf{8} & \mathbf{9} & \mathbf{7} & & \\ & 8 & \mathrm{x} & & 5 & & \\ 5 & 9 & & \mathrm{x} & 15 & 6 & \\ & 7 & 5 & 15 & \mathrm{x} & 8 & 9 \\ & & & 6 & 8 & \mathrm{x} & 11 \\ & & & & 9 & 11 & \mathrm{x} \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} 5 \\ 9 \\ \\ 0 \\ 15 \\ 6 \\ \\ \end{pmatrix}}_{Ax}$$



$A^2 x$: shortest distances within *two* hops:

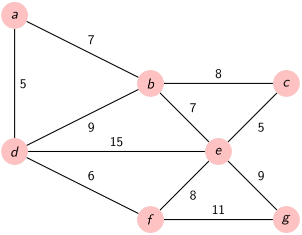
- **swap** input and output vectors, repeat procedure;
- multiple paths reach e.g. b: $\min\{5 + 7 = 12, 9, 15 + 7 = 22\} = 9$.

Example

Single-source shortest-paths (SSSP) using a min-plus semiring:

- graph represented by its $n \times n$ adjacency matrix A
- SSSP via $A^k x$ using the semiring $(N_0, \min, +, \infty, 0)$:

$$\underbrace{\begin{pmatrix} 5 \\ \mathbf{9} \\ 17 \\ {\color{blue}0} \\ 14 \\ 6 \\ 17 \end{pmatrix}}_{Ax += A^3x} += \underbrace{\begin{pmatrix} x & 7 & & 5 & & & \\ 7 & x & 8 & 9 & 7 & & \\ & 8 & x & & 5 & & \\ & 9 & & x & 15 & 6 & \\ & 7 & 5 & 15 & x & 8 & 9 \\ & & & 6 & 8 & x & 11 \\ & & & & 9 & 11 & x \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} 5 \\ 9 \\ 17 \\ 0 \\ 14 \\ 6 \\ 17 \end{pmatrix}}_{A^2x}$$



A^3x: shortest distances within three hops:

- **swap** input and output vectors, repeat procedure;
- output equals input vector; SSSP solved, **terminate**.

GraphBLAS

A non-exclusive list of graph algorithms expressed using GraphBLAS:

- strongly connected components,
- maximal independent set,
- betweenness centrality,
- *k*-core decomposition,
- graph contraction,
- depth-first search,
- triangle counting,
- graph generation,
- graph clustering,
- shortest paths,
- ...and more– see `graphblas.org` for an up-to-date overview.

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one!

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one!

Use different backends **without ever changing the ALP programs**

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one!

Use different backends **without ever changing the ALP programs**

Selecting the **sequential** **auto-vectorising** backend:

```
grbcxx -o myProgram myProgram.cpp
grbrun ./myProgram datasets/west0497.mtx
```

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one!

Use different backends **without ever changing the ALP programs**

Selecting the **shared-memory parallel** **auto-vectorising** backend:

```
grbcxx --backend reference_omp -o myProgram myProgram.cpp
grbrun -b reference_omp ./myProgram datasets/west0497.mtx
```

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one!

Use different backends **without ever changing the ALP programs**

Selecting the **1D distributed-memory parallel** backend (4 nodes):

```
grbcxx -b bsp1d -o myProgram myProgram.cpp
grbrun -b bsp1d -np 4 ./myProgram datasets/west0497.mtx
```

Backends

Compile-time selected **backends**:

1) specific backend for specific architectures or systems;
2) specific backends for specific use cases.

Backends may be **composed** to

- support heterogeneous targets;
- combine shared-memory parallel with an auto-vectorising backend;
- combine shared-, distributed-memory backends into a hybrid one!

Use different backends **without ever changing the ALP programs**

Selecting the **hybrid shared and dist. parallel** backend (10 nodes):

```
grbcxx -b hybrid -o myProgram myProgram.cpp
grbrun -b hybrid -np 10 ./myProgram datasets/west0497.mtx
```

Performance semantics

Every ALP backend defines **performance semantics**:

- work;

Performance semantics

Every ALP backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;

Performance semantics

Every ALP backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;
- whether system calls such as (de-)allocations may be made.

Performance semantics

Every ALP backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;
- whether system calls such as (de-)allocations may be made.

Performance semantics help

- guide programmers to express the best possible algorithm;

Performance semantics

Every ALP backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;
- whether system calls such as (de-)allocations may be made.

Performance semantics help

- guide programmers to express the best possible algorithm;
- gauge scalability: compute resources vs. problem size;
- expose trade-off opportunities: e.g., speed vs. memory;

Performance semantics

Every ALP backend defines **performance semantics**:

- work;
- memory use;
- operator applications;
- inter-process synchronisation steps;
- data movement between user processes and within a single process;
- whether system calls such as (de-)allocations may be made.

Performance semantics help

- guide programmers to express the best possible algorithm;
- gauge scalability: compute resources vs. problem size;
- expose trade-off opportunities: e.g., speed vs. memory;
- automatic choice of algorithms and backends.

Performance semantics

Every ALP program can be systematically costed:

Primitive	Work	Ops	Data movement	Reductions
setElement(x, y, i)	1	-	1	no
set(x, y)	$\min\{n, nz_x + nz_y\}$	-	$nz_x + nz_y$ or $n + nz_y$	no
clear(x)	nz_x	-	nz_x	no
apply$(z, x, y, \odot/M)$	$\min\{n, nz_x + nz_y\}$	$nz_{x\cap y}$	$2\min\{n, nz_x + nz_y\} + nz_{x\cup y}$	no
foldl$(y, x, \odot/M)$	nz_x	$nz_{x\cap y}$	$2nz_x$	no
foldr$(x, y, \odot/M)$	nz_x	$nz_{x\cap y}$	$2nz_x$	no
foldl$(y, \alpha, \odot/M)$	nz_y	nz_y	nz_y	no
foldr$(\alpha, y, \odot/M)$	nz_y	nz_y	nz_y	no
foldl(α, y, M)	nz_y	nz_y	nz_y	yes
foldr(y, α, M)	nz_y	nz_y	nz_y	yes
mul(z, x, y, R)	$\min\{nz_x, nz_y\}$	$nz_{x\cap y}$	$2\min\{nz_x, nz_y\} + nz_{x\cap y}$	no
dot$(z, x, y, (M, \odot))$	n	$2n$	$2n$	yes
dot(z, x, y, R)	$\min\{nz_x, nz_y\}$	$2 \cdot nz_{x\cap y}$	$2\min\{nz_x, nz_y\}$	yes

Level-1 primitives and their costs, excluding masking. Similar tables exist for level-2 and level-3 primitives.

Ref.: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation by Y., D. Di Nardo, J. M. Nash, and W. J. Suijlen (2020).

Sparse vectors

Sparse vectors:

- ideal: $\mathcal{O}(1)$ query, assign, and iteration.
- **sparse accumulators**: nonzero index **stack** and boolean **array**;
- in parallel: synchronise, combine sparsity structures. **Prefix-sum**.

Gilbert, Moler, and Schreiber, *Sparse matrices in MATLAB: Design and implementation.* SIAM JMAA (1992); Y et al., *A C++ GraphBLAS* (2020). Big-Oh bounds in the classical RAM model.

Sparse vectors

Sparse vectors:

- ideal: $\mathcal{O}(1)$ query, assign, and iteration.
- **sparse accumulators**: nonzero index **stack** and boolean **array**;
- in parallel: synchronise, combine sparsity structures. **Prefix-sum**.

Gilbert, Moler, and Schreiber, *Sparse matrices in MATLAB: Design and implementation*. SIAM JMAA (1992); Y et al., *A C++ GraphBLAS* (2020). Big-Oh bounds in the classical RAM model.

Alternative, tree-based map (`std::map`):

- $\mathcal{O}(\log nz)$ query and assign;
- $\mathcal{O}(1)$ iteration.
- parallelisation: join, intersect (set algebra!)

Currently **not** implemented in ALP/GraphBLAS due to overhead.

Davis, *SuiteSparse::GraphBLAS: Graph algorithms in the language of sparse linear algebra*. ACM TOMS ('19).

Sparse matrices

Sparse matrices of size $m \times n$, nz nonzeroes:

- use $\Theta(nz)$ storage, not $\Theta(mn)$,
- level-2 cost $\sim$ number of nonzeroes touched,
- level-3 cost $\sim$ number of operator applications required.

Many sparse matrix storages exist.

Sparse matrices

Sparse matrices of size $m \times n$, nz nonzeroes:

- use $\Theta(nz)$ storage, not $\Theta(mn)$,
- level-2 cost $\sim$ number of nonzeroes touched,
- level-3 cost $\sim$ number of operator applications required.

Many sparse matrix storages exist.

SotA storage cannot do much better than

$$\sim 2(w_V + w_I)nz + w_A(m+1) \text{ bytes,}$$

Sparse matrices

Sparse matrices of size $m \times n$, nz nonzeroes:

- use $\Theta(nz)$ storage, not $\Theta(mn)$,
- level-2 cost $\sim$ number of nonzeroes touched,
- level-3 cost $\sim$ number of operator applications required.

Many sparse matrix storages exist.

SotA storage cannot do much better than

$$\sim 2(w_V + w_I)nz + w_A(m+1) \text{ bytes},$$

unless (theoretically), bit-level compression.

Sparse matrices

Sparse matrices of size $m \times n$, nz nonzeroes:

- use $\Theta(nz)$ storage, not $\Theta(mn)$,
- level-2 cost $\sim$ number of nonzeroes touched,
- level-3 cost $\sim$ number of operator applications required.

Many sparse matrix storages exist.

SotA storage cannot do much better than

$$\sim 2(w_V + w_I)nz + w_A(m+1) \text{ bytes},$$

unless (theoretically), bit-level compression.

We use **Gustafson's format** (CRS+CCS).

Y. and Roose, *High-level strategies for sparse matrix–vector multiplication*, IEEE TPDS 2014.
Y et al., *A C++ GraphBLAS*, 2020.
Simecek, Langr, Tvrdik. *Minimal quadtree format for compression of sparse matrices storage.* SSNA (2012).

Containers

Data structures are **opaque**; data representations are chosen

fully automatically, hidden from user.

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );`
2) `grb::eWiseMul( s, alpha, v, semiring );`

Blocking execution: the vector s is accessed *twice*

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );`

2) `grb::eWiseMul( s, alpha, v, semiring );`

Blocking execution: the vector s is accessed *twice*; performance ✗

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );`

2) `grb::eWiseMul( s, alpha, v, semiring );`

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Ref.: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation by Y., D. Di Nardo, J. M. Nash, and W. J. Suijlen (2020).

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );`
2) `grb::eWiseMul( s, alpha, v, semiring );`

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Ref.: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation by Y., D. Di Nardo, J. M. Nash, and W. J. Suijlen (2020).

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );`
2) `grb::eWiseMul( s, alpha, v, semiring );`

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Automatic **non-blocking mode** (Mastoras et al., '22):

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );`

2) `grb::eWiseMul( s, alpha, v, semiring );`

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Automatic non-blocking mode (Mastoras et al., '22):

- *lazily* evaluate ALP/GraphBLAS calls, no ALP program changes!

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );`

2) `grb::eWiseMul( s, alpha, v, semiring );`

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

Automatic non-blocking mode (Mastoras et al., '22): humble ✓

- *lazily* evaluate ALP/GraphBLAS calls, no ALP program changes!

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

The nonblocking backend

Suppose we compute $s = r + \alpha v$ over a given `semiring`:

1) `grb::set( s, r );`

2) `grb::eWiseMul( s, alpha, v, semiring );`

Blocking execution: the vector s is accessed *twice*; performance ✗

Manual fusion (Y. et al., '20): performance ✓, not very humble ✗

```
grb::eWiseLambda( [ &s, &r, &alpha, &v, &ring ] (const size_t i) {
    grb::apply( s[ i ], alpha, v[ i ], ring.getMultiplicativeOperator() );
    grb::foldl( s[ i ], r[ i ], ring.getAdditiveOperator() );
}, s, r, v );
```

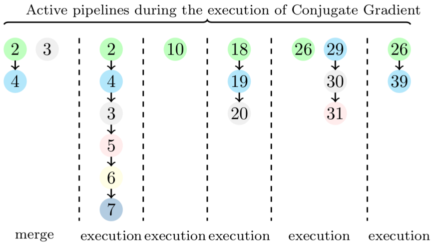
Automatic non-blocking mode (Mastoras et al., '22): humble ✓

- *lazily* evaluate ALP/GraphBLAS calls, no ALP program changes!
- dynamically trigger pipelines when required, **automatically fuse**.

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

The nonblocking backend

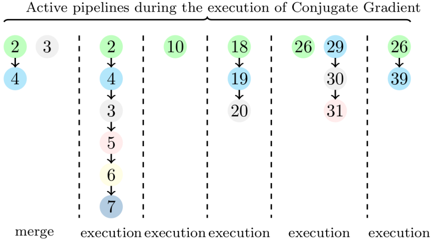
Dynamic **on-line** dependence analysis:




```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

The nonblocking backend

Dynamic **on-line** dependence analysis:



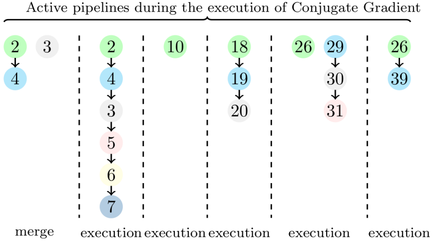
Fused execution can **cross control flow**:

- e.g., lines 26, 39 cross an if-statement;

```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

The nonblocking backend

Dynamic **on-line** dependence analysis:



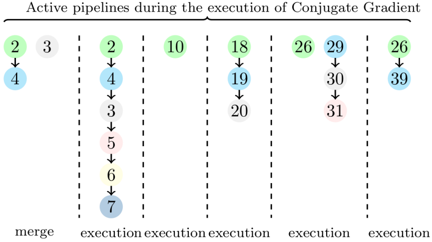

```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

Fused execution can **cross control flow**:

- e.g., lines 26, 39 cross an if-statement;
- elect chunk size s.t. all vectors cached;

The nonblocking backend

Dynamic **on-line** dependence analysis:



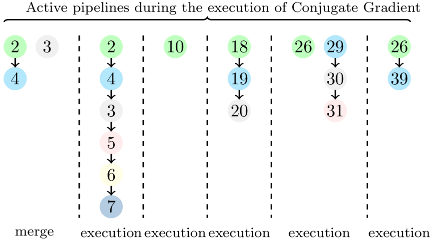
Fused execution can **cross control flow**:

- e.g., lines 26, 39 cross an if-statement;
- elect chunk size s.t. all vectors cached;
- reduce #threads if vectors too small;

```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

The nonblocking backend

Dynamic **on-line** dependence analysis:




```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= sqrt(bnorm);
13
14 iter = 0;
15
16 do {
17     // three-stage pipeline, vectors(temp, u)
18     grb::set(temp, 0);
19     grb::mxv(temp, A, u, ring);
20     grb::dot(residual, temp, u, ring);
21
22     grb::apply(alpha, sigma, residual, divide);
23
24     // part of a two-stage pipeline, vectors (x, u, r)
25     // the eWiseMulAdd at the bottom is the second stage
26     grb::eWiseMulAdd(x, alpha, u, x, ring);
27
28     // three-stage pipeline, vectors(temp, r)
29     grb::eWiseMul(temp, alpha, temp, ring);
30     grb::eWiseApply(r, r, temp, minus);
31     grb::dot(residual, r, r, ring);
32
33     if (sqrt(residual) < tol) break;
34
35     grb::apply(alpha, residual, sigma, divide);
36
37     // part of a two-stage pipeline, vectors (x, u, r)
38     // the eWiseMulAdd aboce is the first stage
39     grb::eWiseMulAdd(u, alpha, u, r, ring);
40
41     sigma = residual;
42 } while (++iter < max_iterations);
```

Fused execution can **cross control flow**:

- e.g., lines 26, 39 cross an if-statement;
- elect chunk size s.t. all vectors cached;
- reduce #threads if vectors too small;
- **analytic model** automatically selects performance parameters: ✓.

Performance

Speedup relative to sequential ALP (v0.5), vs. state-of-the-art

- Conjugate Gradient solve, two-socket x86, 44 cores:

	gyro_m	G2_circuit	bundle_adj	ecology2	Queen_4147
GSL	0.84	0.95	0.89	0.91	0.92
blocking ALP	2.30	4.53	**12.7**	6.91	17.5
SuiteSparse:GraphBLAS	1.57	1.11	5.82	3.52	11.6
Eigen	5.21	2.57	1.61	1.94	9.20
non-blocking ALP	**5.57**	**9.75**	2.87	**13.7**	**18.6**

Ref.: Mastoras, Anagnostidis, and Y., "Nonblocking execution in GraphBLAS", IEEE IPDPSW 2022.
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

Performance

Speedup relative to sequential ALP (v0.5), vs. state-of-the-art

- Conjugate Gradient solve, two-socket x86, 44 cores:

	gyro_m	G2_circuit	bundle_adj	ecology2	Queen_4147
GSL	0.84	0.95	0.89	0.91	0.92
blocking ALP	2.30	4.53	**12.7**	6.91	17.5
SuiteSparse:GraphBLAS	1.57	1.11	5.82	3.52	11.6
Eigen	5.21	2.57	1.61	1.94	9.20
non-blocking ALP	**5.57**	**9.75**	2.87	**13.7**	**18.6**

Our novel nonblocking backend:

- speedup up to pipeline depth if $nz = \Theta(n)$;

Ref.: Mastoras, Anagnostidis, and Y., "Nonblocking execution in GraphBLAS", IEEE IPDPSW 2022.
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

Performance

Speedup relative to sequential ALP (v0.5), vs. state-of-the-art

- Conjugate Gradient solve, two-socket x86, 44 cores:

	gyro_m	G2_circuit	bundle_adj	ecology2	Queen_4147
GSL	0.84	0.95	0.89	0.91	0.92
blocking ALP	2.30	4.53	**12.7**	6.91	17.5
SuiteSparse:GraphBLAS	1.57	1.11	5.82	3.52	11.6
Eigen	5.21	2.57	1.61	1.94	9.20
non-blocking ALP	**5.57**	**9.75**	2.87	**13.7**	**18.6**

Our novel nonblocking backend:

- speedup up to pipeline depth if $nz = \Theta(n)$;
- up to 2.43× vs. blocking, 0.49–8.78× vs. SuiteSparse:GraphBLAS;

Ref.: Mastoras, Anagnostidis, and Y., "Nonblocking execution in GraphBLAS", IEEE IPDPSW 2022.
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

Performance

Speedup relative to sequential ALP (v0.5), vs. state-of-the-art

- Conjugate Gradient solve, two-socket x86, 44 cores:

	gyro_m	G2_circuit	bundle_adj	ecology2	Queen_4147
GSL	0.84	0.95	0.89	0.91	0.92
blocking ALP	2.30	4.53	**12.7**	6.91	17.5
SuiteSparse:GraphBLAS	1.57	1.11	5.82	3.52	11.6
Eigen	5.21	2.57	1.61	1.94	9.20
non-blocking ALP	**5.57**	**9.75**	2.87	**13.7**	**18.6**

Our novel nonblocking backend:

- speedup up to pipeline depth if $nz = \Theta(n)$;
- up to **2.43**$\times$ vs. blocking, 0.49–8.78$\times$ vs. SuiteSparse:GraphBLAS;
- 2.87–**7.06**$\times$ vs. Eigen– which also performs loop fusion.

Ref.: Mastoras, Anagnostidis, and Y., "Nonblocking execution in GraphBLAS", IEEE IPDPSW 2022.
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

Performance

Speedup relative to sequential ALP (v0.5), vs. state-of-the-art

- Conjugate Gradient solve, two-socket x86, 44 cores:

	gyro_m	G2_circuit	bundle_adj	ecology2	Queen_4147
GSL	0.84	0.95	0.89	0.91	0.92
blocking ALP	2.30	4.53	**12.7**	6.91	17.5
SuiteSparse:GraphBLAS	1.57	1.11	5.82	3.52	11.6
Eigen	5.21	2.57	1.61	1.94	9.20
non-blocking ALP	**5.57**	**9.75**	2.87	**13.7**	**18.6**

Our novel nonblocking backend:

- speedup up to pipeline depth if $nz = \Theta(n)$;
- up to **2.43**$\times$ vs. blocking, 0.49–8.78$\times$ vs. SuiteSparse:GraphBLAS;
- 2.87–**7.06**$\times$ vs. Eigen– which also performs loop fusion.

Similar results for PageRank and sparse deep neural network inference.

Ref.: Mastoras, Anagnostidis, and Y., "Nonblocking execution in GraphBLAS", IEEE IPDPSW 2022.
Ref.: —, "Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance", ACM TACO, 2023.

Performance

HPCG benchmark, dual-socket ARM, 96 cores, maximum problem size

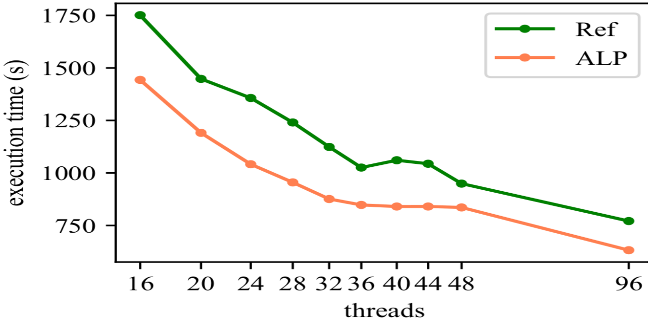
- reference HPCG code modified to use **Red-Black Gauss-Seidel**,
  - ALP cannot express GS; it would not scale.

Performance

HPCG benchmark, dual-socket ARM, 96 cores, maximum problem size

- reference HPCG code modified to use **Red-Black Gauss-Seidel**,
  - ALP cannot express GS; it would not scale.

Comparison, using the blocking ALP backend:



Ref. Scolari, Y.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS", GrAPL at IPDPSW (to appear, 2023)

Performance

Scale-out performance of graph algorithms, using the hybrid backend:

- Clueweb12 link matrix, approx. 978M vertices and 42.5B edges

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	537
4-hop reachability BFS	48.8	110	54.8	99.6	83.0	74.2	23.3
20-hop reachability BFS	404	280	231	323	221	230	160
PageRank	13.3	10.3	9.68	8.00	21.0	22.9	21.6

The k-hop BFS and PageRank (PR) on Clueweb12, performance in seconds. Infiniband EDR interconnect.

Ref.: updated (ALP/GraphBLAS v0.4) results from "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation" by Y. et al. (2020)
Ref.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS" by Scolari & Y., GrAPL at IPDPSW (to appear, 2023)

Performance

Scale-out performance of graph algorithms, using the hybrid backend:

- Clueweb12 link matrix, approx. 978M vertices and 42.5B edges

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	**537**
4-hop reachability BFS	48.8	110	54.8	99.6	83.0	74.2	**23.3**
20-hop reachability BFS	404	280	231	323	221	230	**160**
PageRank	13.3	10.3	9.68	8.00	21.0	22.9	21.6

The k-hop BFS and PageRank (PR) on Clueweb12, performance in seconds. Infiniband EDR interconnect.

Scalable, expected speedup from 4 to 10 nodes is 2.5×:

- parallel I/O: **2.83**×

Ref.: updated (ALP/GraphBLAS v0.4) results from "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation" by Y. et al. (2020)
Ref.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS" by Scolari & Y., GrAPL at IPDPSW (to appear, 2023)

Performance

Scale-out performance of graph algorithms, using the hybrid backend:

- Clueweb12 link matrix, approx. 978M vertices and 42.5B edges

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	537
4-hop reachability BFS	48.8	110	54.8	99.6	83.0	74.2	23.3
20-hop reachability BFS	404	280	231	323	221	230	160
PageRank	13.3	10.3	9.68	8.00	21.0	22.9	21.6

The k-hop BFS and PageRank (PR) on Clueweb12, performance in seconds. Infiniband EDR interconnect.

Scalable, expected speedup from 4 to 10 nodes is 2.5×:

- parallel I/O: **2.83**×; k-hop BFS: 2.09–**2.53**×

Ref.: updated (ALP/GraphBLAS v0.4) results from "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation" by Y. et al. (2020)
Ref.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS" by Scolari & Y., GrAPL at IPDPSW (to appear, 2023)

Performance

Scale-out performance of graph algorithms, using the hybrid backend:

- Clueweb12 link matrix, approx. 978M vertices and 42.5B edges

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	537
4-hop reachability BFS	48.8	110	54.8	99.6	83.0	74.2	23.3
20-hop reachability BFS	404	280	231	323	221	230	160
PageRank	13.3	10.3	9.68	8.00	21.0	22.9	21.6

The k-hop BFS and PageRank (PR) on Clueweb12, performance in seconds. Infiniband EDR interconnect.

Scalable, expected speedup from 4 to 10 nodes is 2.5×:

- parallel I/O: **2.83**×; k-hop BFS: 2.09–**2.53**×; PR: 0.62–**1.66**×.

Ref.: updated (ALP/GraphBLAS v0.4) results from "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation" by Y. et al. (2020)
Ref.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS" by Scolari & Y., GrAPL at IPDPSW (to appear, 2023)

Performance

Scale-out performance of graph algorithms, using the hybrid backend:

- Clueweb12 link matrix, approx. 978M vertices and 42.5B edges

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	537
4-hop reachability BFS	48.8	110	54.8	99.6	83.0	74.2	23.3
20-hop reachability BFS	404	280	231	323	221	230	160
PageRank	13.3	10.3	9.68	8.00	21.0	22.9	21.6

The k-hop BFS and PageRank (PR) on Clueweb12, performance in seconds. Infiniband EDR interconnect.

Scalable, expected speedup from 4 to 10 nodes is 2.5×:

- parallel I/O: **2.83**×; k-hop BFS: 2.09–**2.53**×; PR: 0.62–**1.66**×.

Distributed performance depends on **data distribution**.

Ref.: updated (ALP/GraphBLAS v0.4) results from "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation" by Y. et al. (2020)
Ref.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS" by Scolari & Y., GrAPL at IPDPSW (to appear, 2023)

Performance

Scale-out performance of graph algorithms, using the hybrid backend:

- Clueweb12 link matrix, approx. 978M vertices and 42.5B edges

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	**537**
4-hop reachability BFS	48.8	110	54.8	99.6	83.0	74.2	**23.3**
20-hop reachability BFS	404	280	231	323	221	230	**160**
PageRank	13.3	10.3	9.68	**8.00**	21.0	22.9	21.6

The k-hop BFS and PageRank (PR) on Clueweb12, performance in seconds. Infiniband EDR interconnect.

Scalable, expected speedup from 4 to 10 nodes is $2.5\times$:

- parallel I/O: **2.83**$\times$; k-hop BFS: 2.09–**2.53**$\times$; PR: 0.62–**1.66**$\times$.

Distributed performance depends on **data distribution**.

- distributed-memory backend: row-wise block-cyclic, $T_O = \Theta(n)$

Ref.: updated (ALP/GraphBLAS v0.4) results from "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation" by Y. et al. (2020)
Ref.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS" by Scolari & Y., GrAPL at IPDPSW (to appear, 2023)

Performance

Scale-out performance of graph algorithms, using the hybrid backend:

- Clueweb12 link matrix, approx. 978M vertices and 42.5B edges

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	**537**
4-hop reachability BFS	48.8	110	54.8	99.6	83.0	74.2	**23.3**
20-hop reachability BFS	404	280	231	323	221	230	**160**
PageRank	13.3	10.3	9.68	**8.00**	21.0	22.9	21.6

The k-hop BFS and PageRank (PR) on Clueweb12, performance in seconds. Infiniband EDR interconnect.

Scalable, expected speedup from 4 to 10 nodes is 2.5×:

- parallel I/O: **2.83**×; k-hop BFS: 2.09–**2.53**×; PR: 0.62–**1.66**×.

Distributed performance depends on **data distribution**.

- distributed-memory backend: row-wise block-cyclic, $T_O = \Theta(n)$;
- 2.5D: $\mathcal{O}(n/p^{1/2})$, $\Omega(n/p^{2/3})$.

Ref.: updated (ALP/GraphBLAS v0.4) results from "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation" by Y. et al. (2020)
Ref.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS" by Scolari & Y., GrAPL at IPDPSW (to appear, 2023)

Performance

Scale-out performance of graph algorithms, using the hybrid backend:

- Clueweb12 link matrix, approx. 978M vertices and 42.5B edges

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	**537**
4-hop reachability BFS	48.8	110	54.8	99.6	83.0	74.2	**23.3**
20-hop reachability BFS	404	280	231	323	221	230	**160**
PageRank	13.3	10.3	9.68	**8.00**	21.0	22.9	21.6

The k-hop BFS and PageRank (PR) on Clueweb12, performance in seconds. Infiniband EDR interconnect.

Scalable, expected speedup from 4 to 10 nodes is $2.5\times$:

- parallel I/O: **2.83**$\times$; k-hop BFS: 2.09–**2.53**$\times$; PR: 0.62–**1.66**$\times$.

Distributed performance depends on **data distribution**.

- distributed-memory backend: row-wise block-cyclic, $T_O = \Theta(n)$;
- 2.5D: $\mathcal{O}(n/p^{1/2})$, $\Omega(n/p^{2/3})$. Reference HPCG: $\Theta(n^{1/3}/p^{1/2})$.

Ref.: updated (ALP/GraphBLAS v0.4) results from "A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation" by Y. et al. (2020)
Ref.: "Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS" by Scolari & Y., GrAPL at IPDPSW (to appear, 2023)

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;
- data exchange between Spark and ALP happens within-process.

Performance

ALP **interoperability** with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;
- data exchange between Spark and ALP happens within-process.

Orders of magnitude improvements on **10 nodes** (hybrid backend):

				Spark				Spark with ALP/GraphBLAS			
	GB	Gnz	n_ϵ	$n=1$	$n=10$	$n=n_\epsilon$	s/it.	$n=1$	$n=10$	$n=n_\epsilon$	s/it.
uk-2002	4.7	0.3	73	168.6	1373.8	>4 hrs	133.9	8.7	13.9	48.7	0.56
clueweb12	786	42.5	45	-	-	-	-	658.8	963.2	1875.0	27.7

Pagerank performance in seconds using ten Ivy nodes with Infiniband EDR, Spark 2.3.1, and Hadoop 2.7.7.

- I/O: **19×** faster, computation: **239×** faster for `uk-2002`;

Performance

ALP interoperability with existing (parallel) frameworks

- standard Spark/Scala interface, Spark is **not modified**;
- ALP/GraphBLAS algorithms (here, PageRank) are **not modified**;
- data exchange between Spark and ALP happens within-process.

Orders of magnitude improvements on **10 nodes** (hybrid backend):

	GB	Gnz	n_ϵ	Spark $n=1$	Spark $n=10$	Spark $n=n_\epsilon$	Spark s/it.	Spark with ALP/GraphBLAS $n=1$	Spark with ALP/GraphBLAS $n=10$	Spark with ALP/GraphBLAS $n=n_\epsilon$	Spark with ALP/GraphBLAS s/it.
uk-2002	4.7	0.3	73	168.6	1373.8	>4 hrs	133.9	8.7	13.9	48.7	0.56
clueweb12	786	42.5	45	-	-	-	-	658.8	963.2	1875.0	27.7

Pagerank performance in seconds using ten Ivy nodes with Infiniband EDR, Spark 2.3.1, and Hadoop 2.7.7.

- I/O: **19× faster**, computation: **239× faster** for `uk-2002`;
- Spark Clueweb: out of memory; Blogel (Ammar, Ozsu '18): 128 nodes
  - can handle **141×** larger problems, **12×** fewer resources

Ref.: Suijlen and Y., "Lightweight Parallel Foundations: a model-compliant communication layer", (2019; pre-v0.1 ALP. Results are being refreshed with latest ALP, Scala, Spark, LPF: see ALP/Spark @ GitHub).

Beyond ALP/GraphBLAS

How far can we take this type of programming?

The Alps

The Alps:

- Monte Rosa,
- Matterhorn,
- Weisshorn,
- Jungfrau,
- Rothorn,
- Dom,
- ...

The ALPs

Algebraic Programming

IRs, communication layers, domain-specific languages, libraries and everything in-between for realising Algebraic Programming

Switzerland https://algebraic-programming.gith...

The ALPs:

- ALP/GraphBLAS,
- **ALP/Dense**,
- **ALP/Pregel**,
- ...

The ALPs

Algebraic Programming

IRs, communication layers, domain-specific languages, libraries and everything in-between for realising Algebraic Programming

Switzerland https://algebraic-programming.gith...

The ALPs:

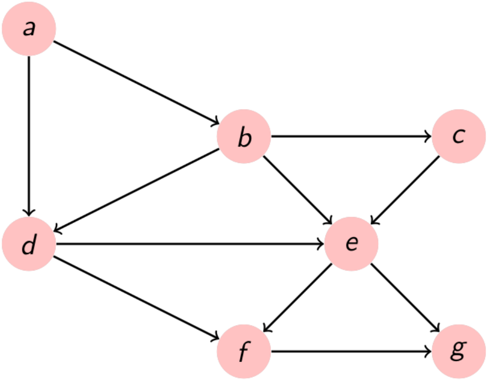
- ALP/GraphBLAS,
- **ALP/Dense**,
- **ALP/Pregel**,
- ...

Interoperability with existing software:

- ALP/Spark;
- ALP/SparseBLAS, ALP/SpBLAS.

ALP/Pregel

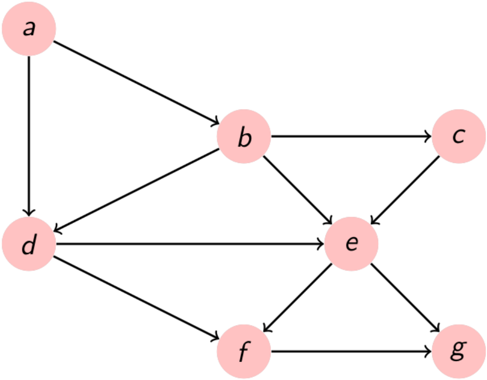
Pregel:



- Each vertex executes a round-based program;
- after each round, message exchange over edges.

ALP/Pregel

Pregel:



- Each vertex executes a round-based program;
- after each round, message exchange over edges.

Think like a vertex, Malewicz et al. '10.

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours;
  - if not, vote to halt program.

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours;
    - if not, vote to halt program.

Pregel **page ranking** over directed graphs:

- start with equally-distributed local score;
- divide it over the number of neighbours and broadcast;
- new score is $\alpha + (1 - \alpha)$ times the sum over incoming scores.

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours;
  - if not, vote to halt program.

Pregel **page ranking** over directed graphs:

- start with equally-distributed local score;
- divide it over the number of neighbours and broadcast;
- new score is $\alpha + (1 - \alpha)$ times the sum over incoming scores.
  - execute a fixed number of rounds, or
  - use local convergence detection and vote-to-halt.

ALP/Pregel

Pregel **connected components** over undirected graphs:

- start with assigning a unique ID;
- broadcast current ID;
- if received any incoming higher ID, replace ours;
    - if not, vote to halt program.

Pregel **page ranking** over directed graphs:

- start with equally-distributed local score;
- divide it over the number of neighbours and broadcast;
- new score is $\alpha + (1 - \alpha)$ times the sum over incoming scores.
    - execute a fixed number of rounds, or
    - use local convergence detection and vote-to-halt.

While “PageRank-like”, *not* mathematically equivalent!

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

- grb :: eWiseLambda executes a vertex program;

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

- grb :: eWiseLambda executes a vertex program;
- grb :: vxm orchestrates message exchange;

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

- grb :: eWiseLambda executes a vertex program;
- grb :: vxm orchestrates message exchange;
- grb :: Monoid performs reductions of incoming messages;

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

- grb :: eWiseLambda executes a vertex program;
- grb :: vxm orchestrates message exchange;
- grb :: Monoid performs reductions of incoming messages;
- grb :: fold reduces halting votes to termination condition.

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

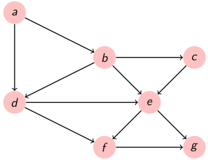
- grb :: eWiseLambda executes a vertex program;
- grb :: vxm orchestrates message exchange;
- grb :: Monoid performs reductions of incoming messages;
- grb :: fold reduces halting votes to termination condition.

```
static void program(
        VertexIDType &current_max_ID,     // each vertex starts with its unique ID
  const VertexIDType &incoming_message,   // IDs will propagate from neighbours
        VertexIDType &outgoing_message,   // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                         // messages arrive after round 1

    if( current_max_ID < incoming_message ) {      // a larger ID has arrived; join the
      current_max_ID = incoming_message;           // component 'led' by this ID

    } else {                                       // otherwise no change: if everyone
      pregel.voteToHalt = true;                    // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;               // as long as we're running, keep
}                                                  // broadcasting my component ID
```



ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

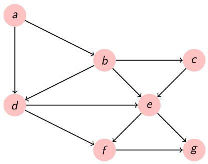
- grb :: eWiseLambda executes a vertex program;
- grb :: vxm orchestrates message exchange;
- grb :: Monoid performs reductions of incoming messages;
- grb :: fold reduces halting votes to termination condition.

```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message,  // IDs will propagate from neighbours
        VertexIDType &outgoing_message,  // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) {     // a larger ID has arrived; join the
      current_max_ID = incoming_message;          // component 'led' by this ID

    } else {                                      // otherwise no change: if everyone
      pregel.voteToHalt = true;                   // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;              // as long as we're running, keep
}                                                 // broadcasting my component ID
```



The translation is **automatic**, using the standard ALP software stack

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

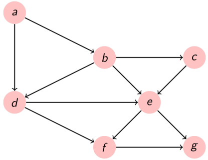
- grb :: eWiseLambda executes a vertex program;
- grb :: vxm orchestrates message exchange;
- grb :: Monoid performs reductions of incoming messages;
- grb :: fold reduces halting votes to termination condition.

```
static void program(
        VertexIDType &current_max_ID,     // each vertex starts with its unique ID
  const VertexIDType &incoming_message,   // IDs will propagate from neighbours
        VertexIDType &outgoing_message,   // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

  if( pregel.round > 0 ) {                        // messages arrive after round 1

    if( current_max_ID < incoming_message ) {     // a larger ID has arrived; join the
      current_max_ID = incoming_message;          // component 'led' by this ID

    } else {                                      // otherwise no change: if everyone
      pregel.voteToHalt = true;                   // has no change, stop execution
    }
  }
  outgoing_message = current_max_ID;              // as long as we're running, keep
}                                                 // broadcasting my component ID
```



The translation is **automatic**, using the standard ALP software stack:

- `grbcxx -b hybrid myPregelAlgo pregelAlgo.cpp`
- `grbrun -b hybrid -np 4 ./myPregelAlgo`

ALP/Pregel

For the Pregel page ranking, **two variants**:

- terminate when all vertices are converged (global);
- disable locally converged vertices from any future rounds (local).

ALP/Pregel

For the Pregel page ranking, **two variants**:

- terminate when all vertices are converged (global);
- disable locally converged vertices from any future rounds (local).

Using masking to not incur overhead from inactive vertices;

- the more deactivated vertices, the faster each compute round.

ALP/Pregel

For the Pregel page ranking, **two variants**:

- terminate when all vertices are converged (global);
- disable locally converged vertices from any future rounds (local).

Using masking to not incur overhead from inactive vertices;

- the more deactivated vertices, the faster each compute round.

Dataset	ALP/Pregel Global	ALP/Pregel Local	Sequential GraphBLAS
gyro_m	34.8 (40)	**24.7** (39)	31.4 (52)
G2_circuit	175 (38)	**78.8** (36)	90.0 (48)
bundle_adj	3 070 (66)	**2 070** (51)	2 330 (60)
G3_circuit	1 960 (38)	**987** (36)	*1 100* (48)
wiki-2007	40 500 (103)	**11 400** (96)	18 100 (55)
uk-2002	153 000 (115)	*46 100* (104)	*72 100* (73)
road_usa	*87 600* (78)	**58 800** (72)	62 200 (78)

Sequential performance in ms. Compares different page ranking algorithms.

ALP/Pregel

Same table, using the blocking shared-memory parallel backend:

Dataset	ALP/Pregel Global	ALP/Pregel Local	Blocking GraphBLAS
gyro_m	31.1 (40)	**29.2** (39)	*37.6* (52)
G2_circuit	58.8 (38)	38.9 (36)	**29.0** (48)
bundle_adj	280 (66)	*224* (51)	1 290 (60)
G3_circuit	367 (38)	243 (36)	**87.8** (48)
wiki-2007	2 440 (103)	**878** (96)	5 030 (55)
uk-2002	11 500 (115)	4 420 (104)	**2 750** (73)
road_usa	9 800 (78)	7 560 (72)	**2 680** (78)

- 0.84–13.0× speedup for the local Pregel page ranking

Ref.: Y., "Humble Heroes", Communications of Huawei Research, to appear (2023).

ALP/Pregel

Same table, using the blocking shared-memory parallel backend:

Dataset	ALP/Pregel Global	ALP/Pregel Local	Blocking GraphBLAS
gyro_m	31.1 (40)	**29.2** (39)	*37.6* (52)
G2_circuit	58.8 (38)	38.9 (36)	**29.0** (48)
bundle_adj	280 (66)	*224* (51)	1 290 (60)
G3_circuit	367 (38)	243 (36)	**87.8** (48)
wiki-2007	2 440 (103)	**878** (96)	5 030 (55)
uk-2002	11 500 (115)	4 420 (104)	**2 750** (73)
road_usa	9 800 (78)	7 560 (72)	**2 680** (78)

- 0.84–13.0× speedup for the local Pregel page ranking;
- fastest 3 out of 7 times (5 out of 13 in the full paper)

Ref.: Y., "Humble Heroes", Communications of Huawei Research, to appear (2023).

ALP/Pregel

Same table, using the blocking shared-memory parallel backend:

Dataset	ALP/Pregel Global	ALP/Pregel Local	Blocking GraphBLAS
gyro_m	31.1 (40)	**29.2** (39)	*37.6* (52)
G2_circuit	58.8 (38)	38.9 (36)	**29.0** (48)
bundle_adj	280 (66)	*224* (51)	1 290 (60)
G3_circuit	367 (38)	243 (36)	**87.8** (48)
wiki-2007	2 440 (103)	**878** (96)	5 030 (55)
uk-2002	11 500 (115)	4 420 (104)	**2 750** (73)
road_usa	9 800 (78)	7 560 (72)	**2 680** (78)

- 0.84–13.0× speedup for the local Pregel page ranking;
- fastest 3 out of 7 times (5 out of 13 in the full paper);
- 1.03–17.5× speedup for connected components algorithm.

Ref.: Y., "Humble Heroes", Communications of Huawei Research, to appear (2023).

ALP/Dense

If generalised sparse linear algebra is so useful, **what about dense**?

Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

ALP/Dense

If generalised sparse linear algebra is so useful, **what about dense**?

- **submatrix selection**, permutations, random sampling, ...

Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

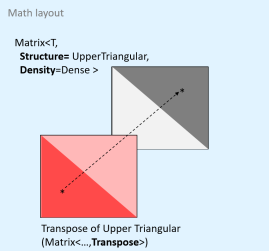
ALP/Dense

If generalised sparse linear algebra is so useful, **what about dense?**

- **submatrix selection**, permutations, random sampling, ...

Requires ALP extensions: structures and views

- structures: general, triangular, banded, ... requires ontology



Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

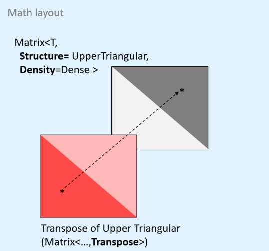
ALP/Dense

If generalised sparse linear algebra is so useful, **what about dense?**

- **submatrix selection**, **permutations**, random sampling, ...

Requires ALP extensions: structures and views

- structures: general, triangular, banded, ... requires ontology
- views: transpose, masks, **permutation**, **submatrix selection**,
  - but also: outer products (two vectors), constant vectors (scalar)



Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

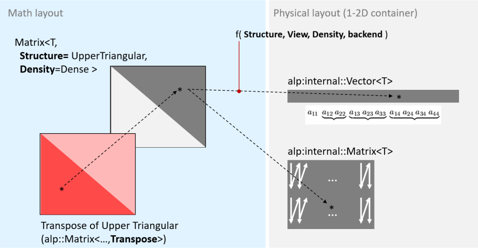
ALP/Dense

If generalised sparse linear algebra is so useful, **what about dense**?

- **submatrix selection**, permutations, random sampling, ...

Requires ALP extensions: structures and views

- structures: general, triangular, banded, ... requires ontology
- views: transpose, masks, **permutation**, **submatrix selection**,
  - but also: outer products (two vectors), constant vectors (scalar)
- opaqueness: ALP controls layout



Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

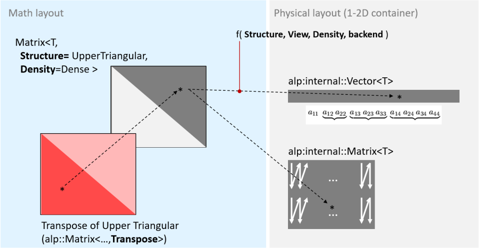
ALP/Dense

If generalised sparse linear algebra is so useful, **what about dense**?

- **submatrix selection**, **permutations**, random sampling, ...

Requires ALP extensions: structures and views

- structures: general, triangular, banded, ... requires ontology
- views: transpose, masks, **permutation**, **submatrix selection**,
  - but also: outer products (two vectors), constant vectors (scalar)
- opaqueness: ALP controls layout, requires parametric **xMFs**



Ref.: Spampinato, Jelovina, Zhuang, Y: Towards Structured Algebraic Programming, ARRAY (2023)

ALP/Dense

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

ALP/Dense

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

ALP/Dense

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

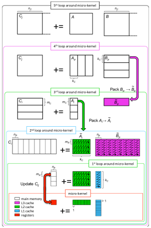
- high-level MLIR dialect introducing, e.g., algebraic structures

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)
2) when pipelines execute, instead first translate to MLIR;
   - high-level MLIR dialect introducing, e.g., algebraic structures
3) BLIS-like approach to optimise MLIR, or dispatch to BLAS:



Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

- high-level MLIR dialect introducing, e.g., algebraic structures

3) BLIS-like approach to optimise MLIR, or dispatch to BLAS:

- use offline (auto-)tuning, **once** per new architecture ✓

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

- high-level MLIR dialect introducing, e.g., algebraic structures

3) BLIS-like approach to optimise MLIR, or dispatch to BLAS:

- use offline (auto-)tuning, **once** per new architecture ✓

4) threads and/or processes execute compiled modules;

- data distribution and parallelisation controlled by ALP.

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

- high-level MLIR dialect introducing, e.g., algebraic structures

3) BLIS-like approach to optimise MLIR, or dispatch to BLAS:

- use offline (auto-)tuning, **once** per new architecture ✓

4) threads and/or processes execute compiled modules;

- data distribution and parallelisation controlled by ALP.

Between JIT and AOT: **delayed compilation** of 'universal binaries'

- high-level MLIR as an **architecture-agnostic** representation

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP/Dense

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)

2) when pipelines execute, instead first translate to MLIR;

- high-level MLIR dialect introducing, e.g., algebraic structures

3) BLIS-like approach to optimise MLIR, or dispatch to BLAS:

- use offline (auto-)tuning, **once** per new architecture ✓

4) threads and/or processes execute compiled modules;

- data distribution and parallelisation controlled by ALP.

Between JIT and AOT: **delayed compilation** of ‘universal binaries’

- high-level MLIR as an **architecture-agnostic** representation,
- can be generated at run-time, following dynamic user control flow.

Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel, while

- Pregel may be viewed as more humble than ALP.

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel, while

- Pregel may be viewed as more humble than ALP.

This observations inspires questions

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel, while

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

- what other humble APIs can ALP efficiently simulate?

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel, while

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

- what other humble APIs can ALP efficiently simulate?
- what other algebras to support for widened applicability?

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel, while

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

- what other humble APIs can ALP efficiently simulate?
- what other algebras to support for widened applicability?
- what are fundamental limitations, how to overcome them?

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel, while

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

- what other humble APIs can ALP efficiently simulate?
- what other algebras to support for widened applicability?
- what are fundamental limitations, how to overcome them?

One software stack with

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel, while

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

- what other humble APIs can ALP efficiently simulate?
- what other algebras to support for widened applicability?
- what are fundamental limitations, how to overcome them?

One software stack with

multiple **humble interfaces**

ALP as a foundational programming model

ALP/Pregel is implemented on top of ALP, thus inheriting

- performance semantics, algebraic type checking, interoperability;
- **automation**: parallelisation, vectorisation, push/pull, ...
- ...and does so at neglible overheads.

ALP is a *more foundational* programming model than Pregel, while

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

- what other humble APIs can ALP efficiently simulate?
- what other algebras to support for widened applicability?
- what are fundamental limitations, how to overcome them?

One software stack with

multiple **humble interfaces**,

achieving **hero performance**.

It's open!

Open source, Apache 2.0, welcome to try, use, and collaborate!

- `https://github.com/Algebraic-Programming`
- `https://algebraic-programming.github.io`

Publications:

- Suijlen, Y.: Lightweight Parallel Foundations: a model-compliant communication layer (2019);
- Y., Di Nardo, Nash, Suijlen: A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation (2020);
- Mastoras, Anagnostidis, Y.: Nonblocking execution in GraphBLAS, IPDPSW (2022);
- Chelini, Barthels, Bientinesi, Copic, Grosser, Spampinato: MOM: Matrix Operations in MLIR, HiPEAC IMPACT workshop (2022);
- Y.: Humble Heroes, Communications of Huawei Research (2023, to appear);
- Mastoras, Anagnostidis, Y.: Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance, ACM TACO (2023);
- Scolari, Y.: Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS, GrAPL at IPDPSW (2023, to appear);
- Spampinato, Jelovina, Zhuang, Y.: Towards Structured Algebraic Programming, ACM ARRAY (2023);
- Papp, Anegg, Y.: Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications, ACM SPAA (2023);
- Papp, Anegg, Y.: DAG scheduling in the BSP model (preprint, 2023);
- Pasadakis, et al., Nonlinear spectral clustering with C++ GraphBLAS, extended abstract, IEEE HPEC (2023, **outstanding short paper**);
- Papp, Anegg, Karanasiou, Y.: Efficient Multi-Processor Scheduling in Increasingly Realistic Models (under preparation, 2023).

Backup slides

Backup slides

Performance

Sparse Deep Neural Network inference, vs. sequential ALP

- GraphChallenge model & data

#layers	#neurons	GSL	Eigen	SS:GrB	non-blocking ALP
1920	64k	0.81×	7.91×	3.98×	**10.8**×
1920	1k	0.64×	0.86×	0.73×	1.39×

Performance

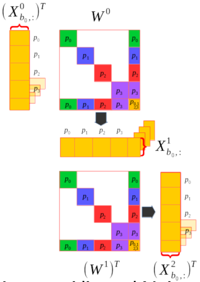
Sparse Deep Neural Network inference, vs. sequential ALP

- GraphChallenge model & data

#layers	#neurons	GSL	Eigen	SS:GrB	non-blocking ALP
1920	64k	0.81×	7.91×	3.98×	**10.8**×
1920	1k	0.64×	0.86×	0.73×	1.39×

Jointly **partition** sparse layers, then tile across layers:

- work by Filip Pawłowski with Uçar and Bisseling
- 5 layers, 64k n.: **1.94**× speedup vs. data-parallel
- 2020 MIT/IEEE GraphChallenge innovation award
- combine with non-blocking ALP/GraphBLAS?



Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).
Ref.: Combinatorial Tiling for Sparse Neural Networks by F. Pawłowski, R. H. Bisseling, B. Uçar, Y. in 2020 IEEE High Performance Extreme Computing Conference (IEEE HPEC)

Performance semantics

Every container has **memory use semantics**:

- "static" costs proportional to container sizes;
- "dynamic" costs proportional to container capacities.

Performance semantics

Every container has **memory use semantics**:

- "static" costs proportional to container sizes;
- "dynamic" costs proportional to container capacities.

Capacities are **optional** during container construction:

```
grb::Vector< bool > s( n, 1 );
grb::Matrix< void > L( n, n, nz );
```

Out of memory errors throw exceptions; primitives return error codes.

Performance semantics

Every container has **memory use semantics**:

- "static" costs proportional to container sizes;
- "dynamic" costs proportional to container capacities.

Capacities are **optional** during container construction:

```
grb::Vector< bool > s( n, 1 );
grb::Matrix< void > L( n, n, nz );
```

Out of memory errors throw exceptions; primitives return error codes.

Capacities:

- are **lower bounds**; grb::capacity(s) ≥ 1;
- may **increase** through grb::resize, updates memory use semantics;
- Any request to decrease capacity thus **may be ignored**.

Basics

User I/O:

```
buildMatrixUnique( A, begin_iterator, end_iterator,
  SEQUENTIAL );
buildMatrixUnique( L, i_begin, i_end, j_begin, j_end,
  PARALLEL );
std::cout << "s␣has␣" << grb::nnz( s ) << "␣values:\n";
for( auto &element : s ){ std::cout << element << "\n"; }
```

Basics

User I/O:

```
buildMatrixUnique( A, begin_iterator, end_iterator,
  SEQUENTIAL );
buildMatrixUnique( L, i_begin, i_end, j_begin, j_end,
  PARALLEL );
std::cout << "s␣has␣" << grb::nnz( s ) << "␣values:\n";
for( auto &element : s ){ std::cout << element << "\n"; }
```

I/O through STL **iterators**:

- input forward iterators (at minimum) over AoS or SoA containers;
- random access input iterators can be shared-memory parallelised(!)

Basics

User I/O:

```
buildMatrixUnique( A, begin_iterator, end_iterator,
  SEQUENTIAL );
buildMatrixUnique( L, i_begin, i_end, j_begin, j_end,
  PARALLEL );
std::cout << "s␣has␣" << grb::nnz( s ) << "␣values:\n";
for( auto &element : s ){ std::cout << element << "\n"; }
```

I/O through STL **iterators**:

- input forward iterators (at minimum) over AoS or SoA containers;
- random access input iterators can be shared-memory parallelised(!)

User processes: each iterator pair on different processes point to

- the same, complete collection C, leading to **sequential I/O**;
- mutually disjoint collections C_i s.t. $C = \cup_i C_i$: **parallel I/O**.

Performance

Auto-vectorisation versus hand-written code, sequential backend

- dot product, dot(alpha, x, y, semiring)
- reduce, foldl < dense >(alpha, x, associativeOp)
- FMA, eWiseMulAdd< dense >(z, alpha, x, y, semiring)

	Ivy Bridge			Cascade Lake		
	Dot product	Reduction	FMA	Dot product	Reduction	FMA
Hand-coded	120 (12.4)	106 (7.03)	**199** (11.2)	227 (6.56)	221 (3.37)	**216** (10.3)
ALP/GraphBLAS	120 (12.4)	106 (7.03)	204 (11.0)	**226** (6.59)	**220** (3.39)	217 (10.3)
eWiseLambda	125 (12.0)	131 (5.69)	205 (10.9)	228 (6.54)	226 (3.30)	217 (10.3)

Microbenchmarks evaluating ALP/GraphBLAS auto-vectorsation. Figures are in milliseconds (and Gbyte/s).

Theoretical (peak) throughput and approximate throughput per core:

- 190.7 GByte/s; 10 cores per CPU, two CPUs, 9.54 Gbyte/s/core (Ivy);
- 262.2 GByte/s; 22 cores / CPU, 2 CPUs, 5.96 Gbyte/s/core (Cascade).