Algebraic Programming

for High Performance Auto-Parallelised Solvers

Albert-Jan N. Yzelman Denis Jelovina Aristeidis Mastoras
Alberto Scolari Daniele G. Spampinato

Computing Systems Laboratory, Huawei Research, Zürich




7th of March, MS54, SIAM PP24

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
  - 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, ...
  - 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

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)

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., Suijlen, Di Nardo, Nash (2017)
- **Algebraic Programming** (2021 onwards)
- ...

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**

```
grb::Vector< std::pair< int, double > > pairs( 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**

```
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
```

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;
```

Algebraic type traits: $\text{is\_associative} < \text{OP} >\text{::value}$, is_commutative, ...

- enables auto-parallelisation and other automatic optimisations.

Basics

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

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

Algebraic type traits: is_associative < OP >::value, is_commutative, ...

- enables auto-parallelisation and other automatic optimisations.

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;
```

Algebraic type traits: is_associative< OP >::value, is_commutative, ...

- enables auto-parallelisation and other automatic optimisations.

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

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

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

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

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

- allow input & output **masks**, **descriptors**, and **phase** arguments;

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

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

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

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );          // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                         // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;           // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );              // t = u * L
foldl( t, gamma, add );                              // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                    // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );          // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                         // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;           // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );              // t = u * L
foldl( t, gamma, add );                              // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                    // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Easy to use: very close to MATLAB, Octave, Eigen, etc.

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );        // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                      // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;        // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );           // t = u * L
foldl( t, gamma, add );                           // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                 // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Easy to use: very close to MATLAB, Octave, Eigen, etc.

- also supports **mixed precision** and **complex** arithmetic:

```
typedef std::complex< double > T; // abstract value type
grb::Vector< T > x;
```

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );        // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                      // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;        // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );           // t = u * L
foldl( t, gamma, add );                           // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                 // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Easy to use: very close to MATLAB, Octave, Eigen, etc.

- also supports **mixed precision** and **complex** arithmetic:

```
typedef std::complex< double > T; // abstract value type
grb::Vector< T > x;
Semiring<
    operators::add< T >, operators::mul< T >,
    identites::zero, identities::one
> plusTimes;
```

Example: PageRank

Inner PageRank loop in ALP, **following a textbook** definition:

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );        // gamma = pr( !r ) * e^T
eWiseApply( u, pr, r, mul );                       // u = pr .* r
gamma = ( alpha * gamma + 1 - alpha ) / n;         // standard scalar arith.
set( t, 0 ); vxm( t, u, L, plusTimes );            // t = u * L
foldl( t, gamma, add );                            // t = t .+ gamma
dot( beta, pr, t, add, absDiff );                  // beta = || pr - t ||_1
std::swap( pr, t ); ++iter;
if( beta <= tol || iter == max_iter ) { break; }
```

Easy to use: very close to MATLAB, Octave, Eigen, etc.

- also supports **mixed precision** and **complex** arithmetic:

```
typedef std::complex< double > T; // abstract value type
grb::Vector< T > x;
Semiring<
    operators::add< T >, operators::mul< T >,
    identites::zero, identities::one
> plusTimes;
dot( beta, x, y, add, operators::conj_right_mul< T >() );
```

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;
- 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
```

The nonblocking backend

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

1) `grb::set( s, r );` // $s = r$
2) `grb::eWiseMul( s, alpha, v, semiring );` // s += $alpha$.* v

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); // $s = r$
2) grb :: eWiseMul(s, alpha, v, semiring); // s += *alpha .∗ v*

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); // $s = r$

2) grb :: eWiseMul(s, alpha, v, semiring); // s += $alpha$.∗ v

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); // $s = r$
2) grb :: eWiseMul(s, alpha, v, semiring); // s += *alpha* .∗ *v*

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); // $s = r$

2) grb::eWiseMul(s, alpha, v, semiring); // s += $alpha .* v$

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); // $s = r$

2) grb :: eWiseMul(s, alpha, v, semiring); // s += $alpha .* v$

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); // $s = r$

2) grb :: eWiseMul(s, alpha, v, semiring); // s += $alpha$.∗ v

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); // $s = r$

2) grb::eWiseMul(s, alpha, v, semiring); // s += $alpha$.∗ v

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 );
```

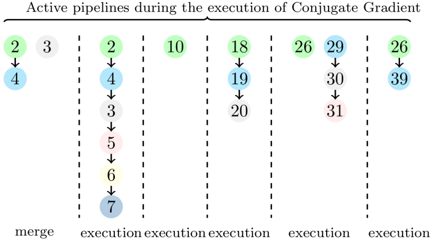
Automatic non-blocking mode (Mastoras et al., '22): humble ✓

- *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

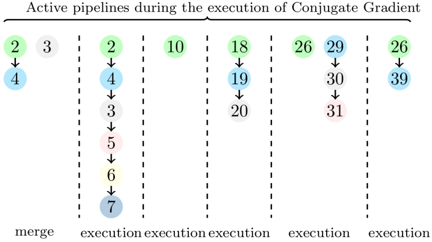
Dynamic **on-line** dependence analysis:




```
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:



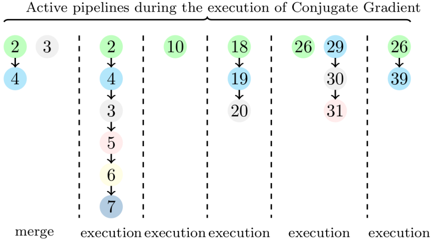
Fused execution can **cross control flow**:

- 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:



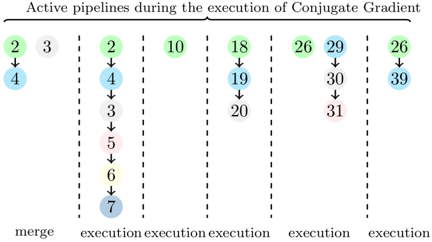
Fused execution can **cross control flow**:

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

```
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:



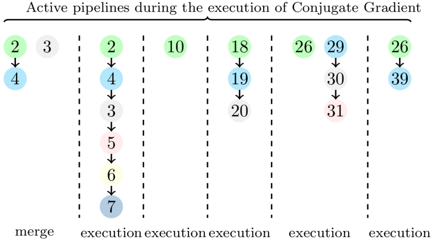

```
1  // six-stage pipeline, vectors(temp, r, x, b, u)
2  grb::set(temp, 0);
3  grb::set(r, 0);
4  grb::mxv(temp, A, x, ring);
5  grb::eWiseApply(r, b, temp, minus);
6  grb::set(u, r);
7  grb::dot(sigma, r, r, ring);
8
9  // single-stage pipeline, vector(b)
10 grb::dot(bnorm, b, b, ring);
11
12 tol *= 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;

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

For the nonblocking backend, expect:

- 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

For the nonblocking backend, expect:

- speedup *up to* pipeline depth if $nz = \Theta(n)$;
- better speedups for higher ratios nz/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

For the nonblocking backend, expect:

- speedup *up to* pipeline depth if $nz = \Theta(n)$;
- better speedups for higher ratios nz/n;
- performance subject to nonzero structure.

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

For the nonblocking backend, expect:

- speedup *up to* pipeline depth if $nz = \Theta(n)$;
- better speedups for higher ratios nz/n;
- performance subject to nonzero structure.

Nonblocking speedup (v0.5), CG solver, 2-socket Cascade (44 cores):

- 5–19% faster than GSL (sequential);

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

For the nonblocking backend, expect:

- speedup *up to* pipeline depth if $nz = \Theta(n)$;
- better speedups for higher ratios nz/n;
- performance subject to nonzero structure.

Nonblocking speedup (v0.5), CG solver, 2-socket Cascade (44 cores):

- 5–19% faster than GSL (sequential);
- 0.25–2.43$\times$ vs. blocking ALP/GraphBLAS;
- 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

For the nonblocking backend, expect:

- speedup *up to* pipeline depth if $nz = \Theta(n)$;
- better speedups for higher ratios nz/n;
- performance subject to nonzero structure.

Nonblocking speedup (v0.5), CG solver, 2-socket Cascade (44 cores):

- 5–19% faster than GSL (sequential);
- 0.25–2.43$\times$ vs. blocking ALP/GraphBLAS;
- 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.

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)

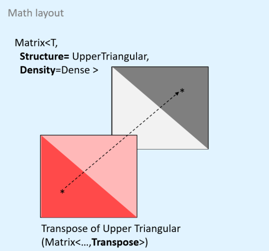
ALP/Dense

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)

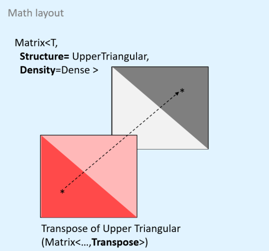
ALP/Dense

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)

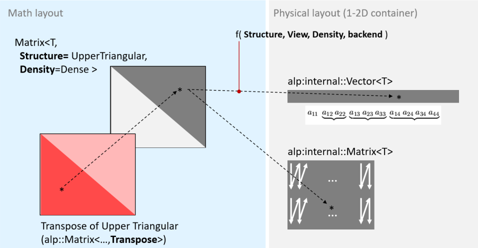
ALP/Dense

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)

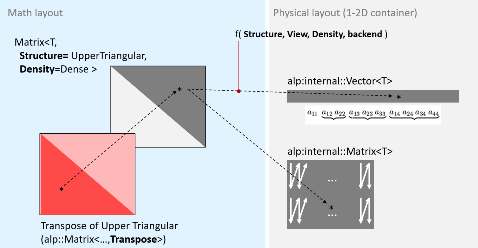
ALP/Dense

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;

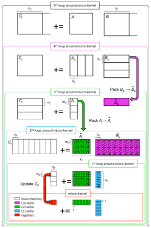
- high-level MLIR dialect introducing, e.g., algebraic structures

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/Dense for solvers

Results:

- permutations, views, structures also useful for sparse

Ref.: Spampinato, Sridhar, Low: Linear algebraic depth-first search, ARRAY (2019).

ALP/Dense for solvers

Results:

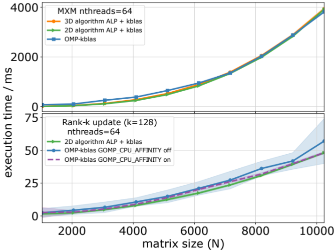
- permutations, views, structures also useful for sparse
- delayed compilation for dense LA: hard *and* unnecessary?

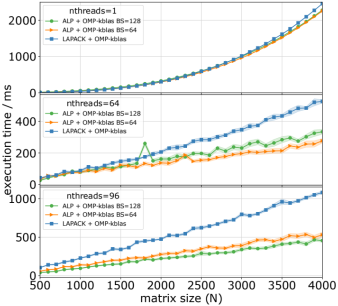
ALP/Dense for solvers

Results:

- permutations, views, structures also useful for sparse
- delayed compilation for dense LA: hard *and* unnecessary?

Dense matrix–matrix multiplication (left) and Cholesky decomposition (right)



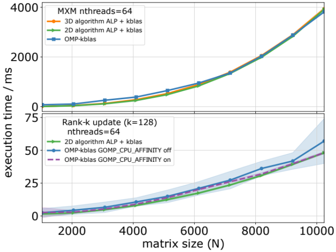


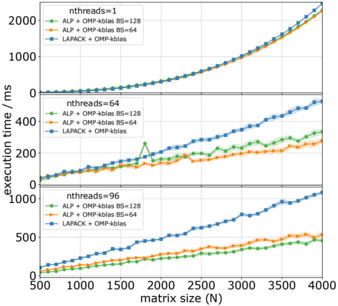
ALP/Dense for solvers

Results:

- permutations, views, structures also useful for sparse
- delayed compilation for dense LA: hard *and* unnecessary?

Dense matrix–matrix multiplication (left) and Cholesky decomposition (right)





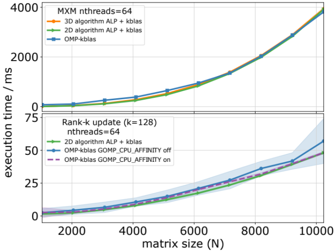
obtained via dispatching to optimised BLAS(!) **Ref.: Spampinato et alii, '23**

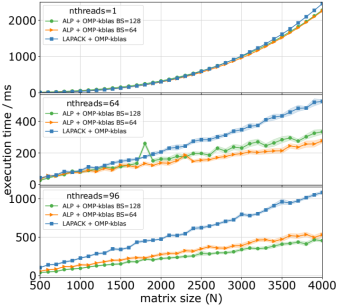
ALP/Dense for solvers

Results:

- permutations, views, structures also useful for sparse
- delayed compilation for dense LA: hard *and* unnecessary?

Dense matrix–matrix multiplication (left) and Cholesky decomposition (right)





obtained via dispatching to optimised BLAS(!) **Ref.: Spampinato et alii, '23**

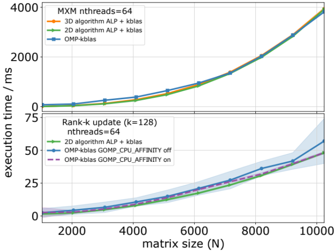
- **nonblocking** ALP/Dense expected to enhance performance;

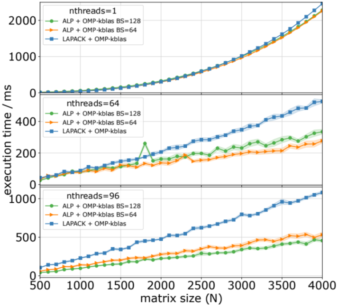
ALP/Dense for solvers

Results:

- permutations, views, structures also useful for sparse
- delayed compilation for dense LA: hard *and* unnecessary?

Dense matrix–matrix multiplication (left) and Cholesky decomposition (right)





obtained via dispatching to optimised BLAS(!) **Ref.: Spampinato et alii, '23**

- **nonblocking** ALP/Dense expected to enhance performance;
- for sparse LA, hard-coded fusion in CG usually only **minor impact**.

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also `http://albert-jan.yzelman.net/alp/v0.8-preview/`)

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also http://albert-jan.yzelman.net/alp/v0.8-preview/)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
              grb::Vector< IOType >         &x,
        const grb::Matrix< NonzeroType > &A,
        const grb::Vector< InputType >    &b,
```

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also http://albert-jan.yzelman.net/alp/v0.8-preview/)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
          grb::Vector< IOType >       &x,
    const grb::Matrix< NonzeroType > &A,
    const grb::Vector< InputType >    &b,
    const std::function< grb::RC(
        grb::Vector< IOType > &,
        const grb::Vector< IOType > &
    ) > &Minv,
```

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also `http://albert-jan.yzelman.net/alp/v0.8-preview/`)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
            grb::Vector< IOType >         &x,
      const grb::Matrix< NonzeroType > &A,
      const grb::Vector< InputType >    &b,
      const std::function< grb::RC(
          grb::Vector< IOType > &,
          const grb::Vector< IOType > &
      ) > &Minv,
      const size_t max_iterations, ResidualType tol,
```

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also http://albert-jan.yzelman.net/alp/v0.8-preview/)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
              grb::Vector< IOType >        &x,
        const grb::Matrix< NonzeroType > &A,
        const grb::Vector< InputType >    &b,
        const std::function< grb::RC(
            grb::Vector< IOType > &,
            const grb::Vector< IOType > &
        ) > &Minv,
        const size_t max_iterations, ResidualType tol,
              size_t &iterations,    ResidualType &residual,
```

Sparse solvers in ALP

ALP v0.8 comes with CG, GMRES, and BiCGstab

- **ease of programming**: BiCGstab in 1h15m
  - including debugging, verified CI testing, and error handling;
  - easily extended, welcoming contributions.
- complex support, explicit *and* user-defined preconditioning.

Interface: (see also http://albert-jan.yzelman.net/alp/v0.8-preview/)

```
template< grb::Descriptor descriptor >
grb::RC preconditioned_conjugate_gradient(
              grb::Vector< IOType >          &x,
      const   grb::Matrix< NonzeroType > &A,
      const   grb::Vector< InputType >   &b,
      const   std::function< grb::RC(
          grb::Vector< IOType > &,
          const grb::Vector< IOType > &
      ) > &Minv,
      const   size_t max_iterations,   ResidualType tol,
              size_t &iterations,      ResidualType &residual,
      grb::Vector< IOType > &buffer1,  grb::Vector< IOType > &buffer2,
      grb::Vector< IOType > &buffer3,  grb::Vector< IOType > &buffer4
);
```

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

ALP v0.8 comes with the following prototypes:

- Sparse BLAS (Duff et al. '02, NIST BLAS tech. forum standard)
- SpBLAS (preceding non-compliant POC, de-facto standard)

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

ALP v0.8 comes with the following prototypes:

- Sparse BLAS (Duff et al. '02, NIST BLAS tech. forum standard)
- SpBLAS (preceding non-compliant POC, de-facto standard)
  - disadvantage: loses some ALP automatisms and performance

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

ALP v0.8 comes with the following prototypes:

- Sparse BLAS (Duff et al. '02, NIST BLAS tech. forum standard)
- SpBLAS (preceding non-compliant POC, de-facto standard)
  - disadvantage: loses some ALP automatisms and performance
- CRS-compatible sparse iterative solvers:

```
double *x, *b, *a_vals; int *a_cols, *a_offs;
// ...

sparse_cg_handle_t handle;
sparse_cg_init( &handle, n, a_vals, a_cols, a_offs );

sparse_cg_solve( handle, x, b );
sparse_cg_destroy( handle );
```

Transition path

Transition paths ensure transparant compatibility:

- ALP generates standard libraries, user programs simply re-link;
- no code changes required on the user side.

ALP v0.8 comes with the following prototypes:

- Sparse BLAS (Duff et al. '02, NIST BLAS tech. forum standard)
- SpBLAS (preceding non-compliant POC, de-facto standard)
  - disadvantage: loses some ALP automatisms and performance
- CRS-compatible solvers with user-defined preconditioning:

```
double *x, *b, *a_vals; int *a_cols, *a_offs;
int my_preconditioner( double * out, const double * in, void * data );
// ...
sparse_cg_handle_t handle;
sparse_cg_init( &handle, n, a_vals, a_cols, a_offs );
sparse_cg_set_preconditioner( handle, my_preconditioner, data );
sparse_cg_solve( handle, x, b );
sparse_cg_destroy( handle );
```

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1
ALP	0.691	0.483	1.95	5.29	7.86	36.0

CG: transition and embrace results, v0.8 (prelim.)

Transition vs. embrace path performance on 2-socket ARM, 96 cores:

- non-preconditioned CG, 1000 iterations, not converged;
  - vectorisation: 16 bytes, nz-indices: `uint`, non-blocking: L1.
- average time over ten runs if sample stddev $< 1\%$;
  - minimum time otherwise, marked red.
- in order: embrace, SparseBLAS transition, and CRS-based trans.

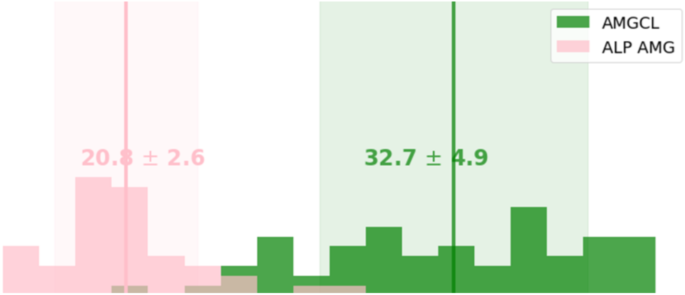
ms./iter	ecology2	apache2	G3circuit	Emilia923	Serena	Queen4147
Eigen	15.1	10.5	22.3	20.0	28.1	99.2
ALP blk.	1.67	1.53	2.45	5.93	8.65	36.4
ALP nb	0.635	0.364	1.94	5.19	7.77	35.7
Opt.	5.19	3.58	7.28	8.84	12.7	50.5
ALP	0.746	0.676	1.91	4.44	6.96	32.9
Opt.	1.25	0.972	2.69	5.21	8.19	39.1
ALP	0.691	0.483	1.95	5.29	7.86	36.0

ALP up to $28.5\times$ faster vs. Eigen and 6.96, $2.01\times$ vs. hand-optimised.

ALP PCG with AMG preconditioning

AMG PCG in ALP vs. AMGCL on an internal problem:

- Matrix size n approx. $217M$, 460M nonzeroes.
- explicit coarsening / restriction matrices applied within ALP;
- compiled using the non-blocking backend.



(obtained using the 2-socket ARM machine, 96 cores; similar for x86)

ALP PCG with ILU(tresh)

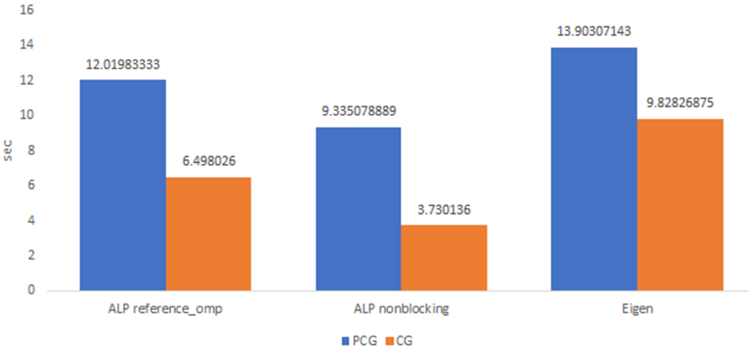
Tresholded ILU PCG in ALP vs. Eigen on the same internal problem:

- forward/backward solves $Lx = b$ and $L^T y = x$ are outside of ALP;
  - optimised using Sympiler/HDAGG (yesterday's plenary)
- they are called from the standard ALP PCG algorithm.

ALP PCG with ILU(tresh)

Tresholded ILU PCG in ALP vs. Eigen on the same internal problem:

- forward/backward solves $Lx = b$ and $L^T y = x$ are outside of ALP;
  - optimised using Sympiler/HDAGG (yesterday's plenary)
- they are called from the standard ALP PCG algorithm.



(obtained on a 2-socket Intel x86 machine, 32 threads)

ALP as a foundational programming model

The ALPs:

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

ALP as a foundational programming model

The ALPs:

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

Integrates with existing software:

- interoperability: ALP/Spark;
- transition paths: ALP/SparseBLAS, ALP/SpBLAS, ALP/Solver.

ALP as a foundational programming model

The ALPs:

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

Integrates with existing software:

- interoperability: ALP/Spark;
- transition paths: ALP/SparseBLAS, ALP/SpBLAS, ALP/Solver.

Future:

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

ALP as a foundational programming model

The ALPs:

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

Integrates with existing software:

- interoperability: ALP/Spark;
- transition paths: ALP/SparseBLAS, ALP/SpBLAS, ALP/Solver.

Future:

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

One software stack, many humble interfaces, hero performance

Ref.: Y., "Humble Heroes", Communications of Huawei Research.

It's open!

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

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

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, IPDPSW (GrAPL 2023);
- 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 (submitted, 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 (submitted, 2023).

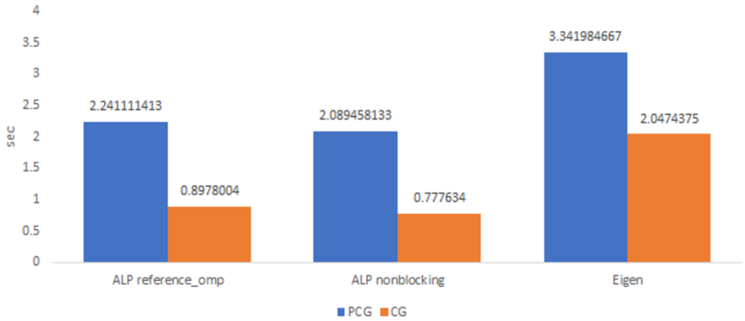
Backup slides

Backup slides

ILU(tresh) PCG on x86

Two-socket Intel x86, smaller problem

- Matrix size *n* approx. 44M, 100M nonzeroes.



(obtained on a 2-socket Intel x86 machine, 32 threads)

Here, PCG Eigen: 3.36, Eigen+Hdagg: 2.08, ALP+Hdagg: 0.778 s.

Libraries generated by ALP

ALP generates the following libs:

- sparseblas and sparseblas_shmem_parallel
- alp_cspblas and alp_cspblas_shmem_parallel
- spsolver and spsolver_shmem_parallel

Libraries generated by ALP

ALP generates the following libs:

- sparseblas and sparseblas_shmem_parallel
- alp_cspblas and alp_cspblas_shmem_parallel
- spsolver and spsolver_shmem_parallel

Generated from standard ALP core primitives and ALP algorithms

- ALP backend implementations and algorithms remain unchanged;

Libraries generated by ALP

ALP generates the following libs:

- sparseblas and sparseblas_shmem_parallel
- alp_cspblas and alp_cspblas_shmem_parallel
- spsolver and spsolver_shmem_parallel

Generated from standard ALP core primitives and ALP algorithms

- ALP backend implementations and algorithms remain unchanged;
- via auto-vectorising serial and nonblocking parallel backends.

Libraries generated by ALP

ALP generates the following libs:

- sparseblas and sparseblas_shmem_parallel
- alp_cspblas and alp_cspblas_shmem_parallel
- spsolver and spsolver_shmem_parallel

Generated from standard ALP core primitives and ALP algorithms

- ALP backend implementations and algorithms remain unchanged;
- via auto-vectorising serial and nonblocking parallel backends.

Key enabler: a **native interface**

```
double *x_raw, *a; int *ja, *ia; size_t m, n;
```

Libraries generated by ALP

ALP generates the following libs:

- sparseblas and sparseblas_shmem_parallel
- alp_cspblas and alp_cspblas_shmem_parallel
- spsolver and spsolver_shmem_parallel

Generated from standard ALP core primitives and ALP algorithms

- ALP backend implementations and algorithms remain unchanged;
- via auto-vectorising serial and nonblocking parallel backends.

Key enabler: a **native interface**

```
double *x_raw, *a; int *ja, *ia; size_t m, n;

grb::Vector< double > x = wrapRawVector< double >( m, x_raw );
grb::Matrix< double > A = wrapCRSMatrix( a, ja, ia, m, n );
```

Libraries generated by ALP

ALP generates the following libs:

- sparseblas and sparseblas_shmem_parallel
- alp_cspblas and alp_cspblas_shmem_parallel
- spsolver and spsolver_shmem_parallel

Generated from standard ALP core primitives and ALP algorithms

- ALP backend implementations and algorithms remain unchanged;
- via auto-vectorising serial and nonblocking parallel backends.

Key enabler: a **native interface**

```
double *x_raw, *a; int *ja, *ia; size_t m, n;

grb::Vector< double > x = wrapRawVector< double >( m, x_raw );
grb::Matrix< double > A = wrapCRSMatrix( a, ja, ia, m, n );

// ...ALP computations...
```

Libraries generated by ALP

ALP generates the following libs:

- sparseblas and sparseblas_shmem_parallel
- alp_cspblas and alp_cspblas_shmem_parallel
- spsolver and spsolver_shmem_parallel

Generated from standard ALP core primitives and ALP algorithms

- ALP backend implementations and algorithms remain unchanged;
- via auto-vectorising serial and nonblocking parallel backends.

Key enabler: a **native interface**

```
double *x_raw, *a; int *ja, *ia; size_t m, n;

grb::Vector< double > x = wrapRawVector< double >( m, x_raw );
grb::Matrix< double > A = wrapCRSMatrix( a, ja, ia, m, n );

// ...ALP computations...

grb::wait( A, x );
```

Libraries generated by ALP

ALP generates the following libs:

- sparseblas and sparseblas_shmem_parallel
- alp_cspblas and alp_cspblas_shmem_parallel
- spsolver and spsolver_shmem_parallel

Generated from standard ALP core primitives and ALP algorithms

- ALP backend implementations and algorithms remain unchanged;
- via auto-vectorising serial and nonblocking parallel backends.

Key enabler: a **native interface**

```
double *x_raw, *a; int *ja, *ia; size_t m, n;

grb::Vector< double > x = wrapRawVector< double >( m, x_raw );
grb::Matrix< double > A = wrapCRSMatrix( a, ja, ia, m, n );

// ...ALP computations...

grb::wait( A, x );

// ...native computations...
```

Performance

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

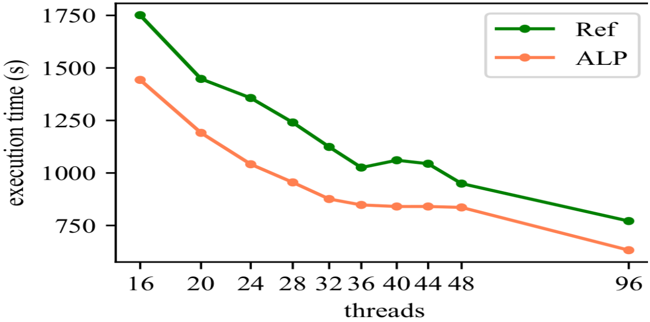
- reference HPCG code modified to use **Red-Black Gauss-Seidel**,
  - ALP cannot express GS; it would not scale.

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 $2.5\times$:

- parallel I/O: $2.83\times$

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 $2.5\times$:

- 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**.

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 $2.5\times$:

- 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: $\mathcal{O}(n/p^{1/2})$, $\Omega(n/p^{2/3})$.

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)

Transition path: CG, x86, ALP v0.5

ms. (and speedup) per CG iteration on x86 (2 CPUs, 88 hyperthreads):

- gyro_m is an extremely small matrix that fits in cache;
- Queen_4147 is around 3.5 GB in-system memory.

Transition path: CG, x86, ALP v0.5

ms. (and speedup) per CG iteration on x86 (2 CPUs, 88 hyperthreads):

- gyro_m is an extremely small matrix that fits in cache;
- Queen_4147 is around 3.5 GB in-system memory.

	gyro_m	vs. Eigen	Queen_4147	vs. SS:GrB
Eigen	0.139	1.00×	77.0	0.79×
GSL	0.864	0.16×	767	0.08×
SS:GrB	0.463	0.30×	61.0	1.00×
ALP/GraphBLAS	0.140	0.99×	39.4	1.55×
ALP/SpBLAS	0.900	0.15×	56.3	1.08×
ALP/SparseBLAS	0.240	0.58×	36.1	1.69×
ALP/Solver	0.131	1.06×	40.2	1.52×

The ALP solvers are up to 1.9× faster than Eigen

Transition path: CG, x86, ALP v0.5

ms. (and speedup) per CG iteration on x86 (2 CPUs, 88 hyperthreads):

- gyro_m is an extremely small matrix that fits in cache;
- Queen_4147 is around 3.5 GB in-system memory.

	gyro_m	vs. Eigen	Queen_4147	vs. SS:GrB
Eigen	0.139	1.00×	77.0	0.79×
GSL	0.864	0.16×	767	0.08×
SS:GrB	0.463	0.30×	61.0	1.00×
ALP/GraphBLAS	0.140	0.99×	39.4	1.55×
ALP/SpBLAS	0.900	0.15×	56.3	1.08×
ALP/SparseBLAS	0.240	0.58×	36.1	1.69×
ALP/Solver	0.131	1.06×	40.2	1.52×

The ALP solvers are up to 1.9× faster than Eigen, and

- up to 1.5× faster than SuiteSparse:GraphBLAS.

Performance semantics

Every backend defines **performance semantics**:

- work;

Performance semantics

Every 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 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 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 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 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$(y, x, \odot/M)$ foldr$(x, y, \odot/M)$	nz_x	$nz_{x\cap y}$	$2nz_x$	no
foldl$(y, \alpha, \odot/M)$ foldr$(\alpha, y, \odot/M)$	nz_y	nz_y	nz_y	no
foldl(α, y, M) 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).

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**.

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; ...