Humble Programming, Revisited

Albert-Jan N. Yzelman

Computing Systems Laboratory, Zürich Research Center, Switzerland

16th of September, 2025

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.

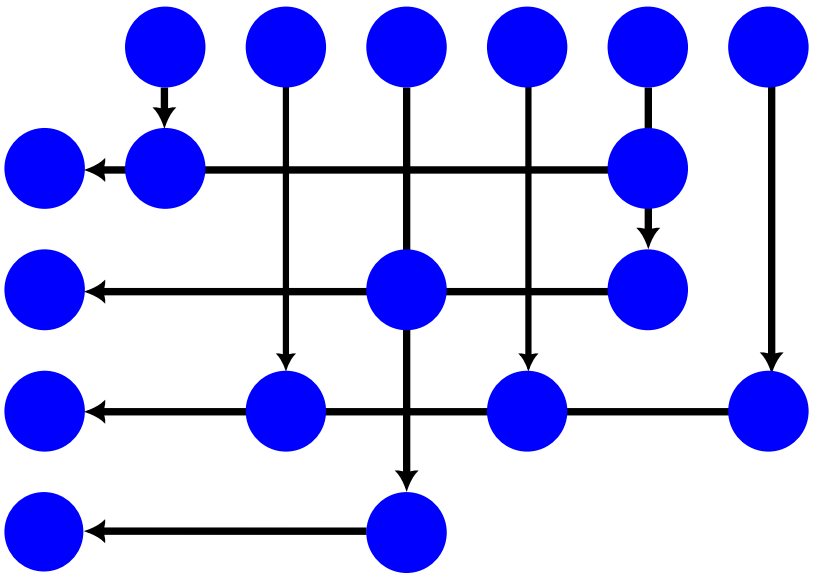
- **Humble** programmers: achieve maximum productivity.
  - easy-to-use, *"scalable"* programming: MapReduce, Spark, ...
  - 100% of peak performance not absolutely required
  - typically **sequential, data-centric, reliable, automatic**

Increasingly many hardware targets,

increasingly heterogeneous hardware:

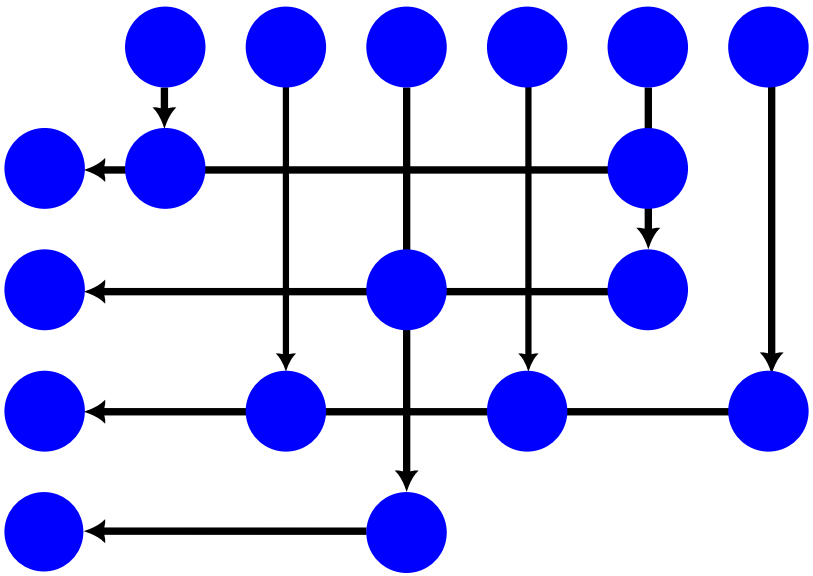
a software productivity crisis is looming

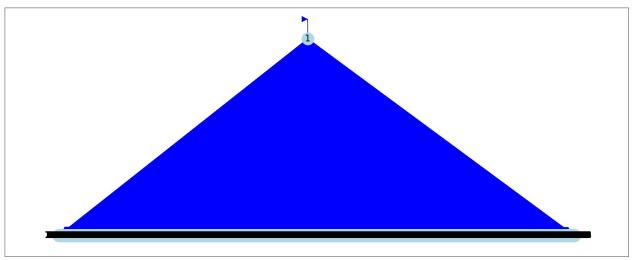
Example: $y = Ax$ with a sparse matrix



Example: $y = Ax$ with a sparse matrix

gyro_m, 17 361 $\times$ 17 361, 340 431 non-zero values:



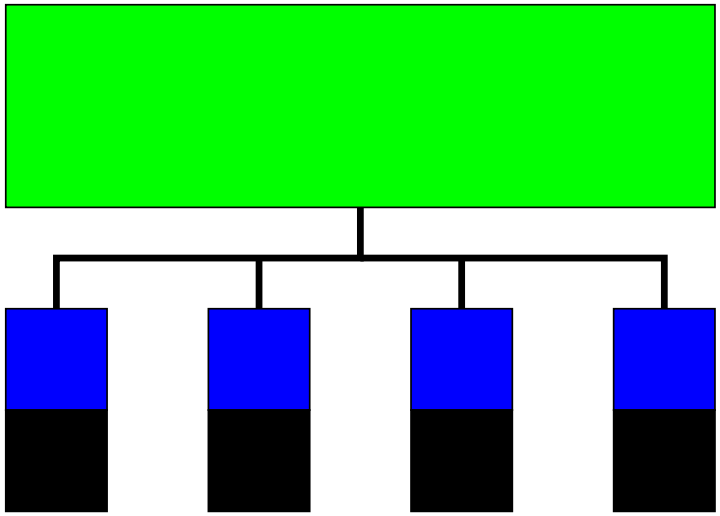


This is a fine-grained DAG representation of the computation:

- fixed to the input, both size & structure;
- **can we schedule this optimally, automatically?**

Scheduling in realistic models

First, BSP computer model & partitioning:



Theorem

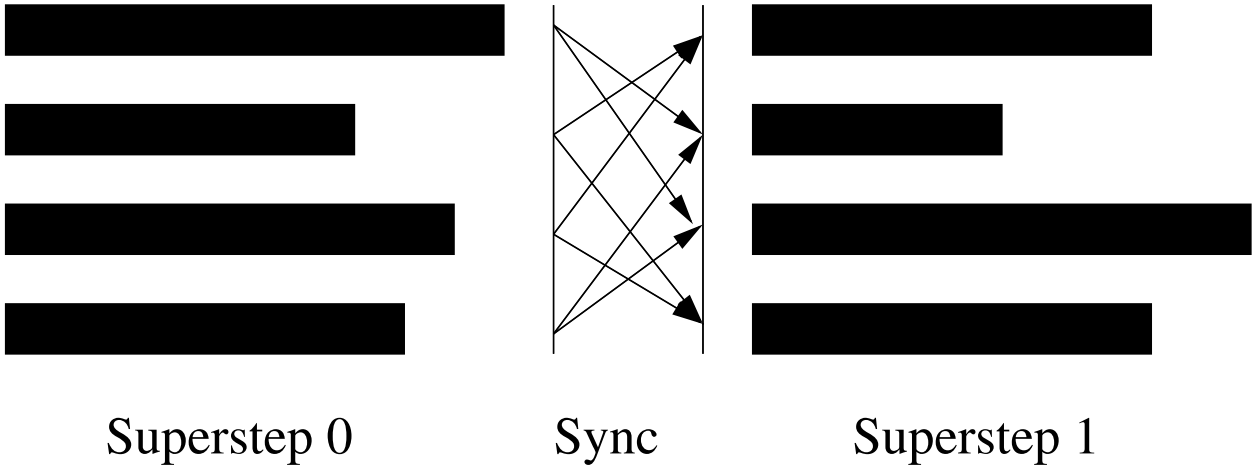
Assuming ETH, it is not possible to approximate the optimum of the ϵ-balanced hypergraph partitioning problem to an $n^{(\log\log n)^{-\delta}}$ factor in polynomial time, for some $\delta > 0$.

This already holds when the hypergraphs are DAGs.

Ref.: Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications, Pál András Papp, Georg Anegg, and Y., SPAA '23 (Theorem 4.1).

Scheduling in realistic models

Second, scheduling via partitioning by imposing BSP supersteps:



i.e., divide DAG into supersteps and then perform layer-wise ϵ-balanced partitioning.

Theorem

It is NP-hard to approximate the layer-wise balanced partitioning problem to any finite factor.

Ref.: Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications, Pál András Papp, Georg Anegg, and Y., SPAA '23 (Theorem 5.2).

Scheduling in realistic models

Third, partition (optimally), then schedule (optimally). Let

- μ be the minimal makespan;
- μ_p be the minimal makespan given a partitioning p.

Goal of post-partitioning phase: $\mu_p \leq (1+\epsilon)\mu$.

Theorem

Given a partitioning, computing its minimal makespan μ_p is NP-hard.

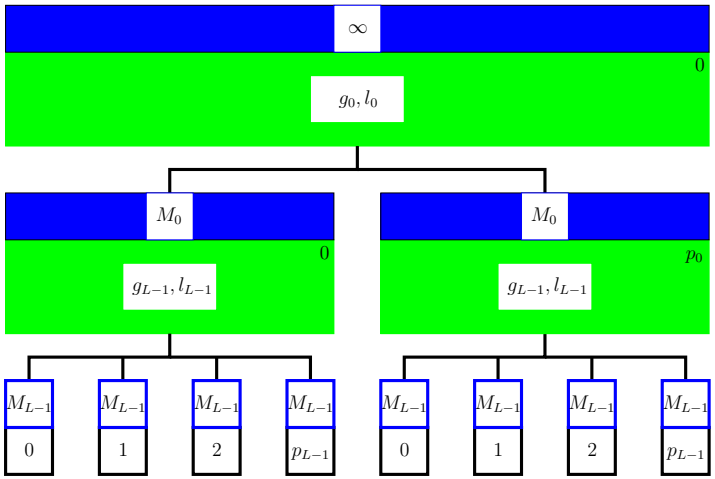
This is already the case for $k = 2$ and out-trees, while

- computing μ can be done in polynomial time in those cases(!)

Ref.: Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications, Pál András Papp, Georg Anegg, and Y., SPAA '23 (Theorem 5.5).

Scheduling in realistic models

Fourth, considering 2-level NUMA (or the Multi-BSP computer model):



Theorem

Given an optimal BSP solution, optimal hierarchical assignment is NP-hard for $p_2 > 2$ and results in a g_1-approximation (up to $\frac{p_1-1}{p_1} g_1$).

(The paper also considers $d > 2$.)

Ref.: Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications, Pál András Papp, Georg Anegg, and Y., SPAA '23 (Theorem 7.{4,5}).

Scheduling in realistic models

Fifth, "pure" BSP scheduling, which is:

Theorem

polynomial-time solvable for chain DAGs and connected chain DAGs.

Theorem

NP-hard already for height-2 DAGs and for in-trees.

Theorem

APX-hard: there is a constant $\epsilon > 0$ s.t. it is NP-hard to approximate to a $(1 + \epsilon)$ factor.

The latter already holds for $p = 2$, and

- holds for several BSP model variations.

Ref.: DAG Scheduling in the BSP Model, Pál András Papp, Georg Anegg, and Y., SOFSEM '25 best paper award. (Theorems 2–5).

Scheduling in realistic models

Sixth, model variations. The optimal schedule in:

Theorem

- *single-port duplex costs at least* $1/2\times$ *the optimum for (sum) BSP;*
- *(sum) BSP costs at least* $(2-\epsilon)\times$ *the optimum for max BSP and single-port duplex;*
- *subset BSP (or* $\alpha-\beta$ *with* $\alpha=0$*) costs at least* $(2-\epsilon)\times$ *the optimum for max BSP and single-port duplex;*
- *max BSP costs at least* $(3/2-\epsilon)\times$ *the optimum for single-port duplex and subset BSP; and*
- *(sum) BSP costs at least* $(3/2-\epsilon)\times$ *the optimum for subset BSP.*

Obtained translating specific DAGs, **disregarding latency costs**.

- in practice, memory or network latencies are significant factors.

Scheduling in realistic models

Sixth, model variations. The optimal schedule in:

Theorem

- *single-port duplex costs at least* $1/2\times$ *the optimum for (sum) BSP;*
- *(sum) BSP costs at least* $(2-\epsilon)\times$ *the optimum for max BSP and single-port duplex;*
- *subset BSP (or* $\alpha-\beta$ *with* $\alpha=0$*) costs at least* $(2-\epsilon)\times$ *the optimum for max BSP and single-port duplex;*
- *max BSP costs at least* $(3/2-\epsilon)\times$ *the optimum for single-port duplex and subset BSP; and*
- *(sum) BSP costs at least* $(3/2-\epsilon)\times$ *the optimum for subset BSP.*

Obtained translating specific DAGs, **disregarding latency costs**.

- in practice, memory or network latencies are significant factors.

One crucial item missing: memory constraints. **Pebbling?**

Ref.: DAG Scheduling in the BSP Model, Pál András Papp, Georg Anegg, and Y., SOFSEM '25 **best paper** award. (Theorems 1, Th. 6 on next slide).
Red-Blue Pebbling with Multiple Processors: Time, Communication and Memory Trade-offs, Toni Böhnlein, Pál András Papp, and Y., SIROCCO '25 (Theorem 1, next slide).
Multiprocessor Scheduling with Memory Constraints: Fundamental Properties and Finding Optimal Solutions, —, ICPP '25 (Theorem 1, next slide).

Scheduling in realistic models

Seventh, towards non-BSP scheduling. For $p > 2$:

Theorem

*BSP **communication scheduling** is NP-hard with free data movement.*

Theorem

Separating multi-processor scheduling from pebbling can return solutions $\Theta(n)$ factors away from optimality.

Scheduling in realistic models

Seventh, towards non-BSP scheduling. For $p > 2$:

Theorem

*BSP **communication scheduling** is NP-hard with free data movement.*

Theorem

Separating multi-processor scheduling from pebbling can return solutions $\Theta(n)$ factors away from optimality.

Theorem

Multi-processor pebbling is APX-hard. (There is a constant $\epsilon > 0$ s.t. it is NP-hard to approximate to a $(1 + \epsilon)$ factor.)

Theorem

*MPP **overhead** is NP-hard to approximate to a finite multiplicative factor, or to any $n^{1-\epsilon}$ additive term (for any $\epsilon > 0$).*

Scheduling in realistic models

Side-track: consider the SpMV multiplication example from earlier:

- each multiplication results in a temporary, same-row sum-reduced.

Do we need all inputs to write out the output vector element?

Theorem

Given a DAG and a fast memory of size r, it is NP-hard to decide whether pebbling with partial computations can reduce costs.

There exists a DAG construction such that optimal pebbling costs $\Theta(n)$ while partial-computation pebbling costs 2.

Theorem

Optimal partial pebbling solutions are NP-hard to approximate to any $n^{1-\epsilon}$ additive term (for any $\epsilon > 0$) or to any multiplicative factor.

Scheduling in realistic models

Side-track: consider the SpMV multiplication example from earlier:

- each multiplication results in a temporary, same-row sum-reduced.

Do we need all inputs to write out the output vector element?

Theorem

Given a DAG and a fast memory of size r, it is NP-hard to decide whether pebbling with partial computations can reduce costs.

There exists a DAG construction such that optimal pebbling costs $\Theta(n)$ while partial-computation pebbling costs 2.

Theorem

Optimal partial pebbling solutions are NP-hard to approximate to any $n^{1-\epsilon}$ additive term (for any $\epsilon > 0$) or to any multiplicative factor.

RBP lower bounds carry over for FFT, GEMM, and FA.

PRBP GEMV: I/O lower bound reduced by $n - 1$.

Ref.: The Impact of Partial Computations on the Red-Blue Pebble Game, Pál András Papp, Aleksandros Sobczyk, and Y., SPAA '25. (Theorems 4.5, 7.1).

Practical scheduling in realistic models

We *can* compute practical **optimal** schedules in realistic models:

- up to four levels, (hyper)DAGs of up to 10 000 nodes:
  - uniform scheduling: 0.56×, 0.76× the cost of Cilk, HDagg;
  - NUMA-aware scheduling: 0.40×, 0.57× vs. Cilk, HDagg;

Practical scheduling in realistic models

We *can* compute practical **optimal** schedules in realistic models:

- up to four levels, (hyper)DAGs of up to 10 000 nodes:
  - uniform scheduling: 0.56×, 0.76× the cost of Cilk, HDagg;
  - NUMA-aware scheduling: 0.40×, 0.57× vs. Cilk, HDagg;
  - multi-level scheduling: close to 0.20× vs. HDagg.

Practical scheduling in realistic models

We *can* compute practical **optimal** schedules in realistic models:

- up to four levels, (hyper)DAGs of up to 10 000 nodes:
  - uniform scheduling: 0.56×, 0.76× the cost of Cilk, HDagg;
  - NUMA-aware scheduling: 0.40×, 0.57× vs. Cilk, HDagg;
  - multi-level scheduling: close to 0.20× vs. HDagg.
- two-level with memory constraints, 40-80 node (hyper)DAGs
  - optimal ILP: 0.77× the cost of two-stage SotA heuristics;
  - optimal ILP: 0.88× the cost of two-stage optimal ILPs;

Practical scheduling in realistic models

We *can* compute practical **optimal** schedules in realistic models:

- up to four levels, (hyper)DAGs of up to 10 000 nodes:
  - uniform scheduling: 0.56×, 0.76× the cost of Cilk, HDagg;
  - NUMA-aware scheduling: 0.40×, 0.57× vs. Cilk, HDagg;
  - multi-level scheduling: close to 0.20× vs. HDagg.
- two-level with memory constraints, 40-80 node (hyper)DAGs
  - optimal ILP: 0.77× the cost of two-stage SotA heuristics;
  - optimal ILP: 0.88× the cost of two-stage optimal ILPs;
- **divide and conquer** allows for 264-464 node (hyper)DAGs:
  - divide-and-conquer ILP: 0.66–1.13× the cost of SotA.

(Problem instances from the HyperDAG DB on the ALP GitHub page.)

Ref.: Efficient Multi-Processor Scheduling in Increasingly Realistic Models, Pál András Papp, Georg Anegg, Aikaterini Karanasiou, and Y., SPAA '24.
Multiprocessor Scheduling with Memory Constraints: Fundamental Properties and Finding Optimal Solutions, Pál András Papp, Toni Böhnlein, and Y., ICPP '25.

Practical scheduling in realistic models

Sparse triangular solve by DAG scheduling: iterate on row-ranges.

Practical scheduling in realistic models

Sparse triangular solve by DAG scheduling: iterate on row-ranges.

Dataset	GL	Funnel+GL	SpMP	HDagg
SuiteSparse	10.8	10.2	7.6	3.3
SuiteSparse (permuted)	15.9	15.4	9.4	9.0
SuiteSparse (ILU(0))	15.1	14.8	8.4	6.8
(synthetic skipped)				

Speedups vs. serial execution on Intel x86_64

Practical scheduling in realistic models

Sparse triangular solve by DAG scheduling: iterate on row-ranges.

Dataset	GL	Funnel+GL	SpMP	HDagg
SuiteSparse	10.8	10.2	7.6	3.3
SuiteSparse (permuted)	15.9	15.4	9.4	9.0
SuiteSparse (ILU(0))	15.1	14.8	8.4	6.8
(synthetic skipped)				

Speedups vs. serial execution on Intel x86_64

- Funnel+GL: 1.14–1.31× less synchronisations(!);

Practical scheduling in realistic models

Sparse triangular solve by DAG scheduling: iterate on row-ranges.

Dataset	GL	Funnel+GL	SpMP	HDagg
SuiteSparse	10.8	10.2	7.6	3.3
SuiteSparse (permuted)	15.9	15.4	9.4	9.0
SuiteSparse (ILU(0))	15.1	14.8	8.4	6.8
(synthetic skipped)				

Speedups vs. serial execution on Intel x86_64

- Funnel+GL: 1.14–1.31× less synchronisations(!);
- on AMD, 1.42× vs. SpMP, 2.63× vs. HDagg;
- on ARM, 4.29× vs. HDagg.

Practical scheduling in realistic models

Sparse triangular solve by DAG scheduling: iterate on row-ranges.

Dataset	GL	Funnel+GL	SpMP	HDagg
SuiteSparse	10.8	10.2	7.6	3.3
SuiteSparse (permuted)	15.9	15.4	9.4	9.0
SuiteSparse (ILU(0))	15.1	14.8	8.4	6.8
(synthetic skipped)				

Speedups vs. serial execution on Intel x86_64

- Funnel+GL: 1.14–1.31× less synchronisations(!);
- on AMD, 1.42× vs. SpMP, 2.63× vs. HDagg;
- on ARM, 4.29× vs. HDagg.

All of this (optimal & heuristics) are available freely, Apache 2.0:

- github.com/Algebraic-Programming/OneStopParallel

Ref.: Efficient Multi-Processor Scheduling in Increasingly Realistic Models, Pál András Papp, Georg Anegg, Aikaterini Karanasiou, and Y., SPAA '24.
Efficient Parallel Scheduling for Sparse Triangular Solvers, Toni Böhnlein, Pál András Papp, Raphael S. Steiner, Christos K. Matzoros, Y., arXiv: 2503.05408; pre-print (June 2025).

Do we need to operate on DAGs?

Back to practice: **do we need all of this**?

No;

Do we need to operate on DAGs?

Back to practice: **do we need all of this**?

No; a well-known alternative are algorithmic **skeletons**:

- heroes can optimise specific skeletons, and, if generic enough;
- frameworks can map humble programs to optimised skeletons.

Do we need to operate on DAGs?

Back to practice: **do we need all of this**?

No; a well-known alternative are algorithmic **skeletons**:

- heroes can optimise specific skeletons, and, if generic enough;
- frameworks can map humble programs to optimised skeletons.

The remainder focuses on "algebraic skeletons", and

- **generalised sparse linear algebra** in particular;
- other skeletons also came by yesterday (e.g., prefix sum).

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;
- compile-time **introspect**ion 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;
- compile-time **introspect**ion of algebraic information;
- auto-**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**.

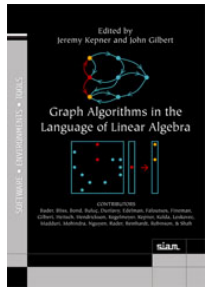
Principles:

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

(ALP/)GraphBLAS

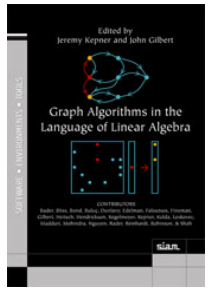
GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

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




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

(ALP/)GraphBLAS



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

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

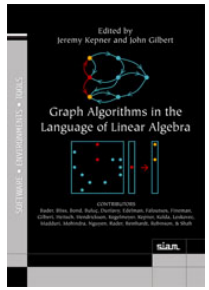
```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
                grb::Vector< VectorT >  &y,
          const grb::Matrix< NonzeroT > &A, const size_t k,
          const grb::Vector< VectorT >  &x,
                grb::Vector< VectorT >  &buffer,
          const Semiring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

Solve different problems with different semirings:

- plus-times: numerical linear algebra,

(ALP/)GraphBLAS

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



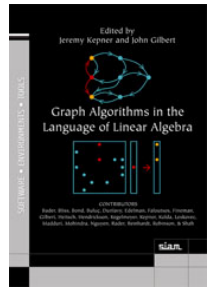
- for example, $y = A^k x$, parametrised in a semiring:

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

Solve different problems with different semirings:

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

(ALP/)GraphBLAS



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

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

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

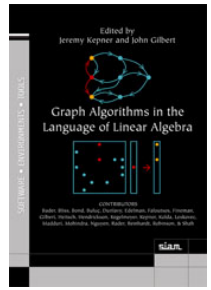
Solve different problems with different semirings:

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

(ALP/)GraphBLAS

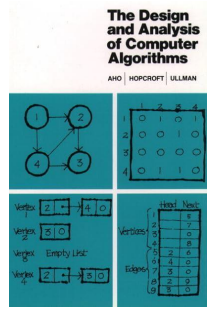
GraphBLAS.org; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

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




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

Solve different problems with different semirings:



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

ALP Backends

Compile-time selected **backends**:

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

ALP 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.

ALP 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**(!)

ALP 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
```

ALP 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
```

ALP 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
```

ALP 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
```

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.
- 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× faster vs. Eigen and 6.96, 2.01× vs. hand-optimised.

Beyond ALP/GraphBLAS

How far can we take this type of programming?

The Alps

The Alps:

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

The ALPs

Algebraic Programming

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

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

The ALPs:

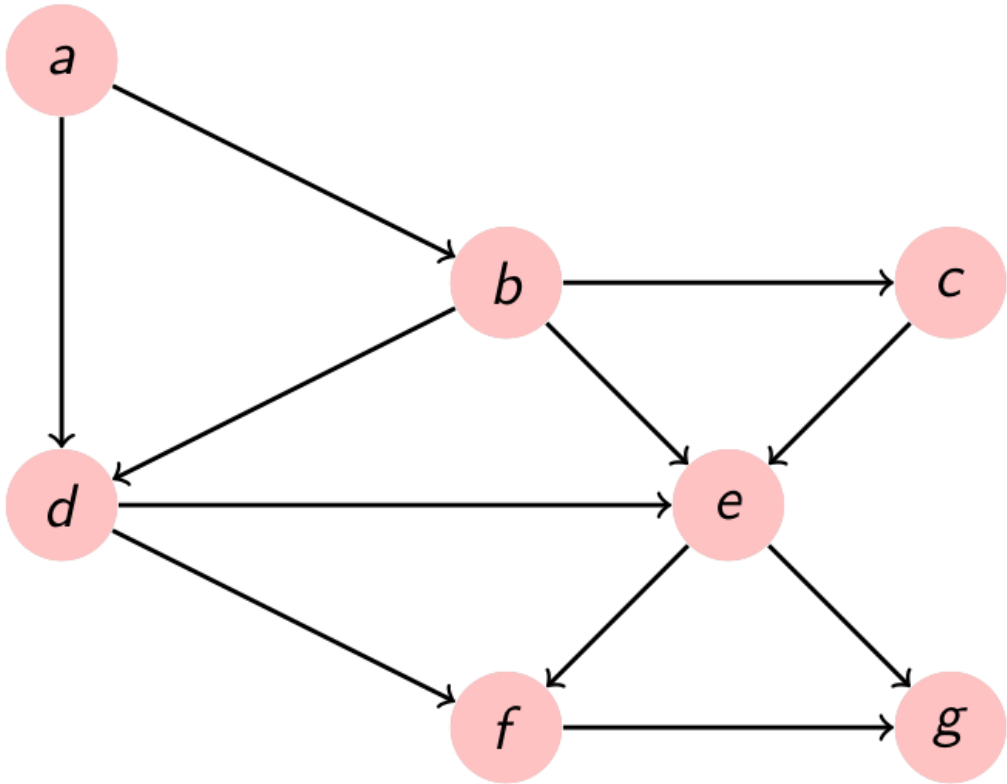
- linear algebra: **ALP/GraphBLAS** and **ALP/Dense**,
- vertex-centric programming: **ALP/Pregel**,
- towards tensor algebra: **ALP/Tensors** and **ALP/Ascend**,
- ...

Interoperability with existing software:

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

ALP/Pregel

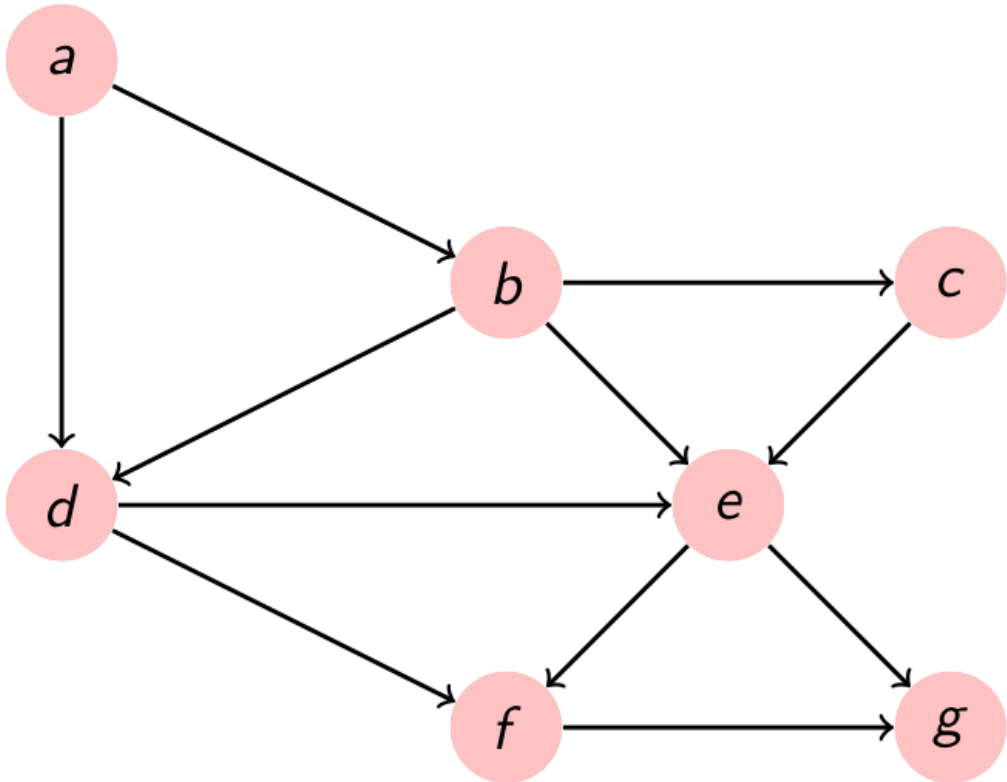
Pregel:



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

ALP/Pregel

Pregel:



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

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

Pregel **page ranking** over directed graphs:

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

Pregel **page ranking** over directed graphs:

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

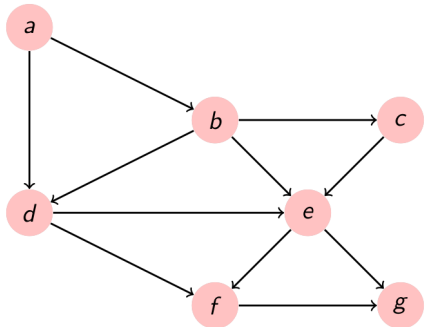
Pregel **page ranking** over directed graphs:

- start with equally-distributed local score;
- divide it over the number of neighbours and broadcast;
- new score is $\alpha + (1 - \alpha)$ times the sum over incoming scores.
  - execute a fixed number of rounds, or
  - use local convergence detection and vote-to-halt.
- while "PageRank-like", **not mathematically equivalent**!

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

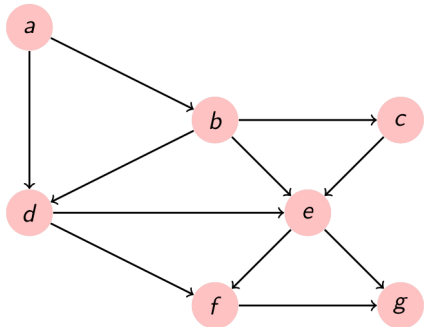
```
static void program(
        VertexIDType &current_max_ID,    // each vertex starts with its unique ID
  const VertexIDType &incoming_message, // IDs will propagate from neighbours
        VertexIDType &outgoing_message, // new max IDs will be broadcast
  grb::interfaces::PregelData &pregel
) {

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

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

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



ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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

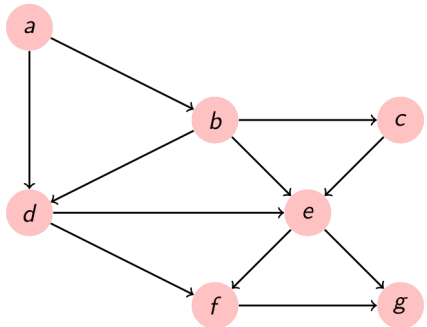


- grb::eWiseLambda executes a vertex program;

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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

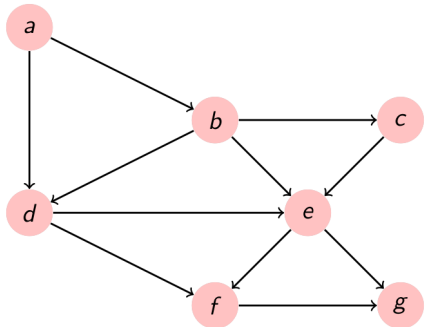


- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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

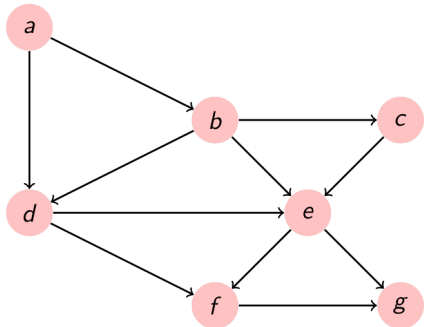


- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;
- grb::Monoid performs reductions of incoming messages;

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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

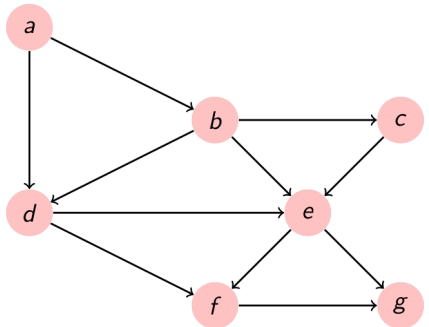


- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;
- grb::Monoid performs reductions of incoming messages;
- grb::fold reduces halting votes to termination condition.

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



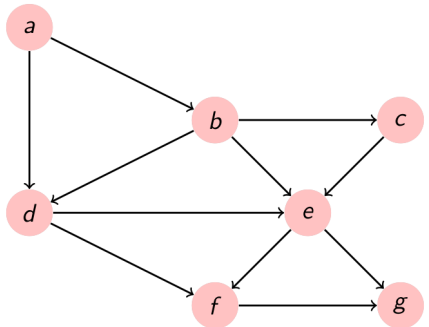
- grb::eWiseLambda executes a vertex program;
- grb::vxm orchestrates message exchange;
- grb::Monoid performs reductions of incoming messages;
- grb::fold reduces halting votes to termination condition.

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

ALP/Pregel

Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



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

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

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

ALP/Pregel

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

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

ALP/Pregel

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

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

Using masking to not incur overhead from inactive vertices;

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

ALP/Pregel

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

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

Using masking to not incur overhead from inactive vertices;

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

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

Sequential performance in ms. Compares different page ranking algorithms.

ALP/Pregel

When using the (blocking) shared-memory parallel backend:

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

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

Ref.: Y., "Humble Heroes", Communications of Huawei Research (2024).

ALP/Pregel

When using the (blocking) shared-memory parallel backend:

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

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

Ref.: Y., "Humble Heroes", Communications of Huawei Research (2024).

ALP as a foundational programming model

The ALPs:

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

ALP as a foundational programming model

The ALPs:

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

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, ALP/Ascend, ...

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, ALP/Ascend, ...

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 (2024).

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 (2020);
- Chelini, Barthels, Bientinesi, Copic, Grosser, Spampinato: MOM: Matrix Operations in MLIR, HiPEAC IMPACT workshop (2022);
- 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);
- Pasadakis, Schenk, Vlaçić, Y.: Nonlinear spectral clustering with C++ GraphBLAS, extended abstract, IEEE HPEC (2023, **outstanding short paper**);
- Niu: Rethinking Sparse Tensor Storage, Master's Thesis, ETHZ (2023);
- Papp, Anegg, Karanasiou, Y.: Efficient Multi-Processor Scheduling in Increasingly Realistic Models, ACM SPAA (2024);
- Y.: Humble Heroes, Communications of Huawei Research (2024);
- Martinez-Ferrer, Y., Beltran: Distributed and heterogeneous tensor-vector contraction algorithms for high performance computing, FGCS (2025);
- Böhnlein, Papp, Steiner, Y.: Efficient Parallel Scheduling for Sparse Triangular Solvers (Brief Summary), IPDPSW (2025);
- Mastoras, Y.: Efficient handling of sparse vectors for parallel nonblocking execution in GraphBLAS, GraphSys at EuroPar (2025).

Backup slides

Backup slides

Performance

ALP interoperability with existing (parallel) frameworks

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

Performance

ALP interoperability with existing (parallel) frameworks

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

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

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

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

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

Performance

ALP interoperability with existing (parallel) frameworks

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

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

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

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

- I/O: 19× faster, computation: 239× faster for `uk-2002`;
  - can handle 141× larger problems, 12× fewer resources

Performance

ALP interoperability with existing (parallel) frameworks

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

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

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

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

- I/O: 19× faster, computation: 239× faster for `uk-2002`;
  - can handle 141× larger problems, 12× fewer resources
- v0.8 (prelim.), **3 nodes**, uk-2002, IB EDR, dual-socket ARM:
  - 113 ms./iter, 19.2× vs. GraphX. See ALP/Spark @ GitHub

Ref.: Suijlen and Y., "Lightweight Parallel Foundations", (2019); ALP v0.8-preview and ALP/Spark (2024).

HPCG: multi-node benchmark mimicking AMG

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

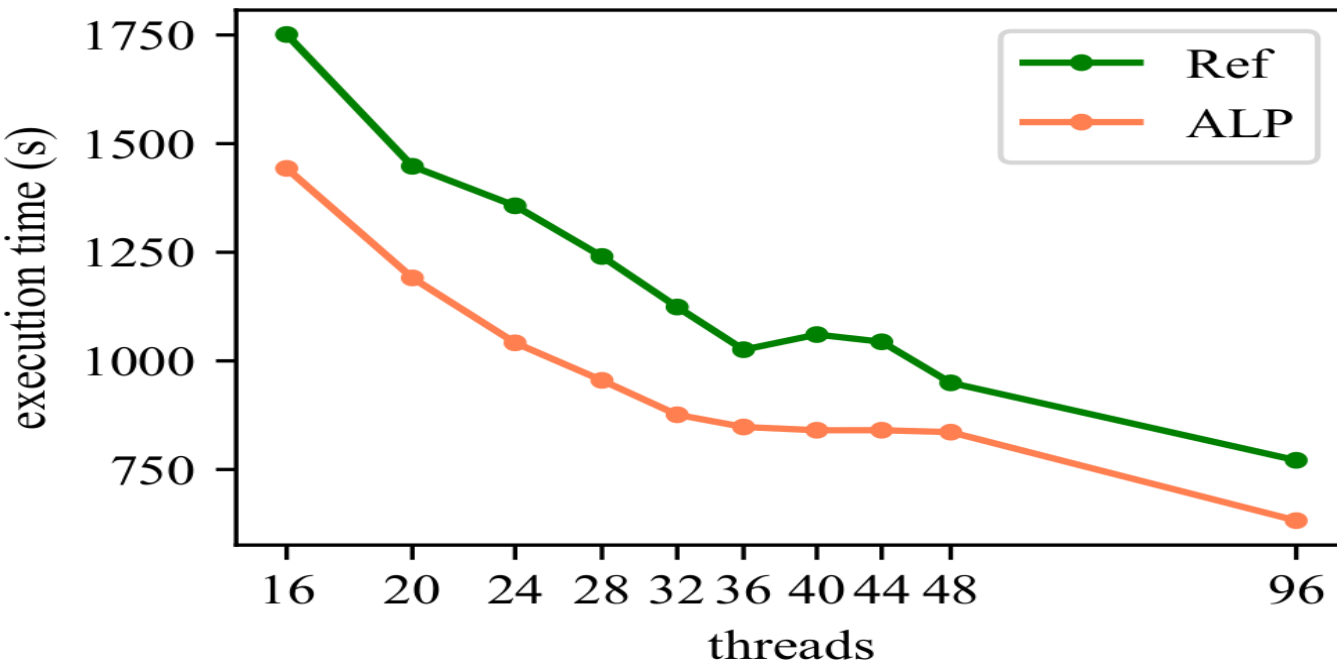
- reference HPCG code modified to use **Red-Black Gauss-Seidel**,
  - ALP cannot express GS; it would not scale.

HPCG: multi-node benchmark mimicking AMG

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

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

Comparison, using the blocking ALP backend:



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

Performance

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

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

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

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

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

Performance

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

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

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

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

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

- parallel I/O: 2.83×

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

Performance

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

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

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

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

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

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

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

Performance

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

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

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

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

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

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

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

Performance

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

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

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

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

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

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

Distributed performance depends on **data distribution**.

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

Performance

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

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

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

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

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

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

Distributed performance depends on **data distribution**.

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

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

Performance

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

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

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

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

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

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

Distributed performance depends on **data distribution**.

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

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

Performance

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

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

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

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

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

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

Distributed performance depends on **data distribution**.

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

Example: PageRank

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

```
beta = gamma = 0;
foldl< invert_mask >( gamma, pr, r, add );          // gamma = sum( pr( !r ) )
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 = sum( pr( !r ) )
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;
grb::semirings::plusTimes< T > plusTimes;
grb::Vector< T > x;
T squaredNorm;
// ...
grb::dot( squaredNorm, x, x, plusTimes );
```

GraphBLAS

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

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

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)
- ...

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 : $D_1 \times D_2 \rightarrow D_3$ reads

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

Basics

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

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

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

- enables auto-parallelisation and other automatic optimisations.

Basics

Algebraic structures are types. E.g., min : $D_1 \times D_2 \rightarrow D_3$ 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 : $D_1 \times D_2 \rightarrow D_3$ 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); // $y \ += Ax$; 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); // $y \ += Ax$; 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.

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

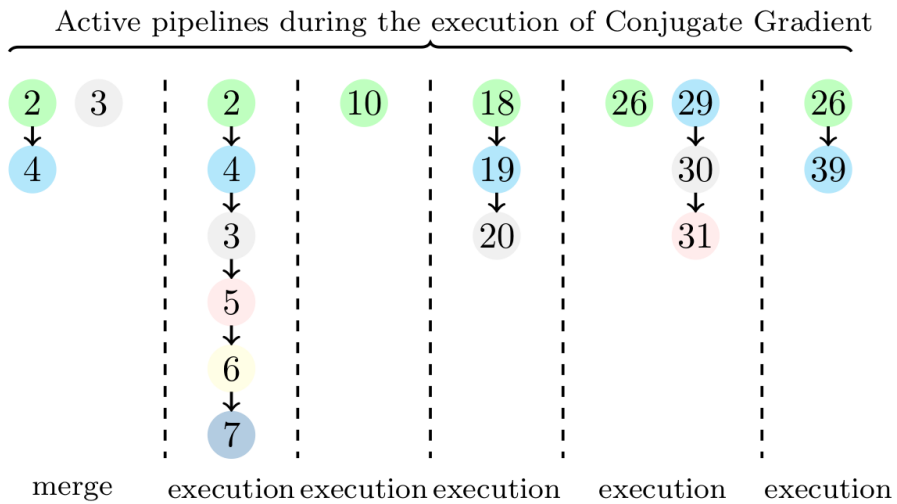
- ALP backend implementations and algorithms remain unchanged;
- via auto-vectorising serial and nonblocking parallel backends.

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...
```

The nonblocking backend

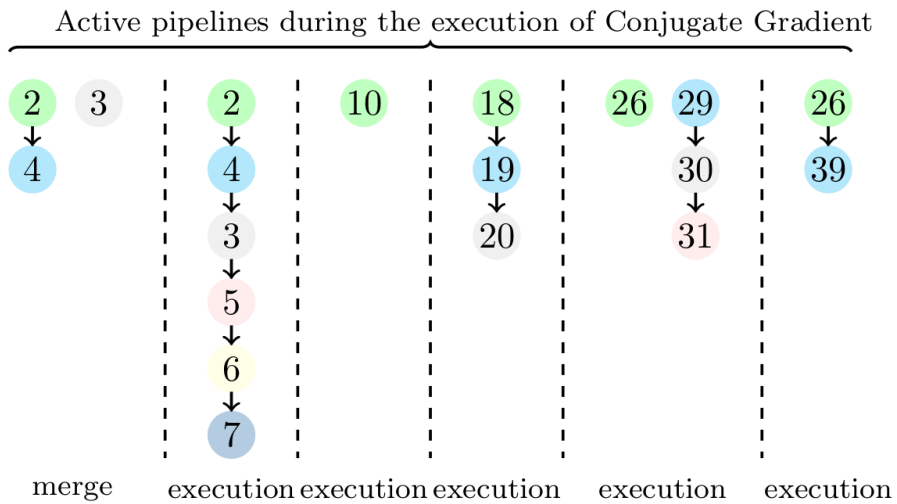
Dynamic **on-line** dependence analysis:




```
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:



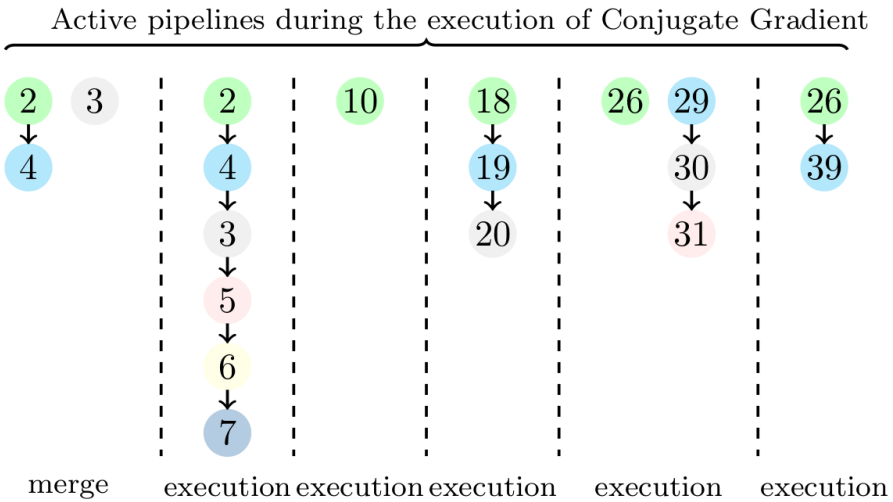
Fused execution can **cross control flow**:

- 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:



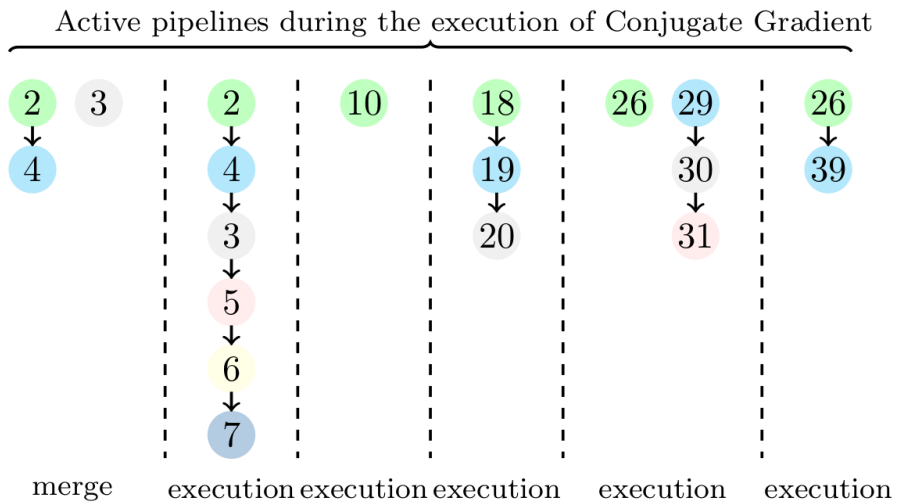
Fused execution can **cross control flow**:

- 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:



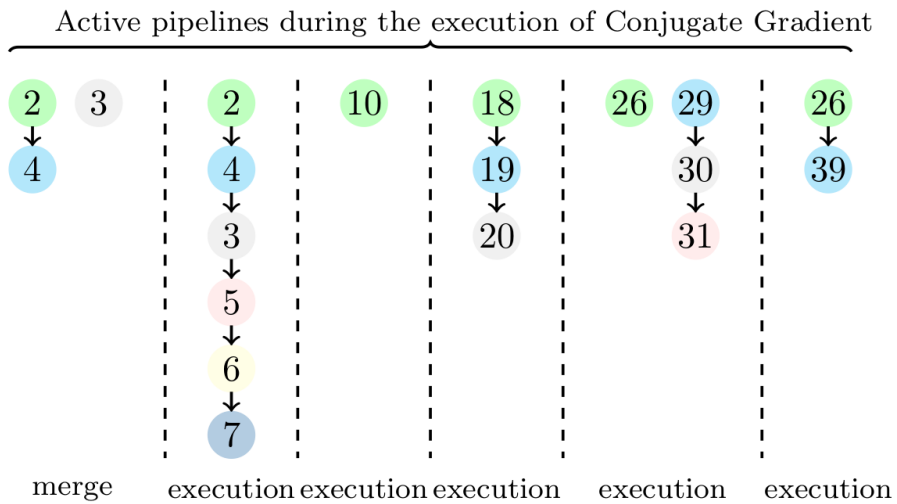
Fused execution can **cross control flow**:

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

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

The nonblocking backend

Dynamic **on-line** dependence analysis:



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: ✓.

```
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);
```

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; ...