Progress in Algebraic Programming and Hypergraphs

Albert-Jan N. Yzelman

Computing Systems Laboratory
Huawei Zürich Research Center

5th of September, 2022

Humble and Hero Programming

- **Hero** programmers: maximum efficiency;

- **Humble** programmers (Dijkstra '74): maximum productivity.

Humble and Hero Programming

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

- **Humble** programmers (Dijkstra '74): maximum productivity.

Humble and Hero Programming

- **Hero** programmers: 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 (Dijkstra '74): maximum productivity.

Humble and Hero Programming

- **Hero** programmers: 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 (Dijkstra '74): maximum productivity.

  - easy-to-use, "*scalable*" programming: MapReduce, Spark, ...

Humble and Hero Programming

- **Hero** programmers: 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 (Dijkstra '74): maximum productivity.
  - easy-to-use, *"scalable"* programming: MapReduce, Spark, ...
  - typically **sequential, data-centric, reliable, & automatic**

Humble and Hero Programming

- **Hero** programmers: 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 (Dijkstra '74): maximum productivity.
  - easy-to-use, *"scalable"* programming: MapReduce, Spark, ...
  - typically **sequential, data-centric, reliable, & automatic**
  - achieving 100% of peak performance not needed

Humble and Hero Programming

- **Hero** programmers: 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 (Dijkstra '74): maximum productivity.
  - easy-to-use, *"scalable"* programming: MapReduce, Spark, ...
  - typically **sequential, data-centric, reliable, & automatic**
  - achieving 100% of peak performance not needed

A central challenge: the **looming programmer productivity crisis**

- increasingly many hardware targets,
- increasingly heterogeneous hardware.

Compilers, libraries, and applications

No one classical solution suffices:

- new hardware, provide standard libraries

Compilers, libraries, and applications

No one classical solution suffices:

- new hardware, provide standard libraries
- compile humble code to novel architectures

Compilers, libraries, and applications

No one classical solution suffices:

- new hardware, provide standard libraries
- compile humble code to novel architectures
- user-driven compilation, DSLs...

Compilers, libraries, and applications

No one classical solution suffices:

- new hardware, provide standard libraries
- compile humble code to novel architectures
- user-driven compilation, DSLs...
- ...would end-users rewrite their application stacks?

Compilers, libraries, and applications

No one classical solution suffices:

- new hardware, provide standard libraries
- compile humble code to novel architectures
- user-driven compilation, DSLs...
- ...would end-users rewrite their application stacks?

Solution: **re-define the classical boundaries** between

- compiler, library, *and* application.

Algebraic Programming

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

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

Algebraic Programming

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

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

Algebraic Programming

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

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

Algebraic Programming

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

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

Principles:

- **explicitly annotate** computations with algebraic information

Algebraic Programming

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

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

Principles:

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

Algebraic Programming

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

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

Principles:

- **explicitly annotate** computations with algebraic information
- allow **compile-time introspection** of algebraic information
- perform **optimisations based on algebraic information**

Algebraic Programming

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

Algebraic Programming

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

For example, in ALP/GraphBLAS:

Algebraic Programming

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

For example, in ALP/GraphBLAS:

1) containers: scalars, vectors, and matrices;

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

Algebraic Programming

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

For example, in ALP/GraphBLAS:

1) containers: scalars, vectors, and matrices;

2) structures: binary operators, monoids, and semirings; and

```
grb::Vector< double > x( n ), y( m ), z( n );
grb::Matrix< void > A( m, n );
grb::operators::min< double > minOp;
grb::operators::add< double, double, double > addOp;
```

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

Algebraic Programming

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

For example, in ALP/GraphBLAS:

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

```
grb::Vector< double > x( n ), y( m ), z( n );
grb::Matrix< void > A( m, n );
grb::operators::min< double > minOp;
grb::operators::add< double, double, double > addOp;

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

grb::set( x, 1.0 );                     // x_i = 1, ∀i
grb::setElement( x, 3.0, n/2 );         // x_{n/2} = 3
grb::eWiseApply( y, x, x, addOp ); // y_i = x_i + x_i, ∀i
grb::eWiseApply( z, x, x, minOp ); // z_i = min{x_i, x_i}, ∀i
grb::mxv( y, A, x, mySemiring );    // y += Ax
```

Algebraic Type Traits

Compile-time inspection of algebraic properties **algebraic type traits**:

- 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

Compile-time inspection of algebraic properties **algebraic type traits**:

- 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 enable:

1) detect programmer errors,

```
grb::Monoid< grb::operators::divide< int >, grb::identities::one > myMonoid;
```

Non-associative monoid? ✗: **compile-time** error, with **clear error message**

Algebraic Type Traits

Compile-time inspection of algebraic properties **algebraic type traits**:

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 enable:

1) detect programmer errors,

```
grb::Monoid< grb::operators::divide< int >, grb::identities::one > myMonoid;
```

Non-associative monoid? ✗: **compile-time** error, with **clear error message**

2) decide which optimisations are applicable,

`grb::eWiseApply( y, x, x, minOp )` with idempotent op? Replace with `grb::set( y, x )` ✓

`grb::mxv( y, A, x, semiring)` has commutative additive monoid: reordered matrix traversal? ✓

Algebraic Type Traits

Compile-time inspection of algebraic properties **algebraic type traits**:

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 enable:

1) detect programmer errors,

`grb::Monoid< grb::operators::divide< int >, grb::identities::one > myMonoid;`
Non-associative monoid? ✗: **compile-time** error, with **clear error message**

2) decide which optimisations are applicable, and

`grb::eWiseApply( y, x, x, minOp )` with idempotent op? Replace with `grb::set( y, x )` ✓
`grb::mxv( y, A, x, semiring)` has commutative additive monoid: reordered matrix traversal? ✓

3) reject expressions without recipe for **auto-parallelisation** exist.

$\alpha = x_0 \odot \ldots \odot x_{n-1}$, i.e., `grb::foldl( alpha, x, op )` with non-associative op? ✗

ALP/GraphBLAS

A **backend**:

- implements the ALP/GraphBLAS API for a **specific system**;

ALP/GraphBLAS

A **backend**:

- implements the ALP/GraphBLAS API for a **specific system**;
- defines **performance semantics**;
  - memory use, work, intra- and inter-process data movement, and inter-process synchronisations;

ALP/GraphBLAS

A **backend**:

- implements the ALP/GraphBLAS API for a **specific system**;
- defines **performance semantics**;
  - memory use, work, intra- and inter-process data movement, and inter-process synchronisations;

Free and open source (Apache 2.0 license, v0.6 since last month):

- `github.com/Algebraic-Programming/ALP`

ALP/GraphBLAS

A **backend**:

- implements the ALP/GraphBLAS API for a **specific system**;
- defines **performance semantics**;
  - memory use, work, intra- and inter-process data movement, and inter-process synchronisations;

Free and open source (Apache 2.0 license, v0.6 since last month):

- `github.com/Algebraic-Programming/ALP`

Current backends:

- sequential auto-vectorising, shared-memory parallel (OpenMP), distributed-memory parallel (LPF+MPI/ibverbs), and hybrid.

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

Performance

Upcoming **nonblocking** CG solve speedup:

- two-socket Intel Cascade Lake, 44 cores, no hyperthreads

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

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

Performance

Upcoming **nonblocking** CG solve speedup:

- two-socket Intel Cascade Lake, 44 cores, no hyperthreads

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

- at most 0.25× slowdown, up to 2.43× faster than blocking ALP;
- at most 0.49× slowdown, up to 8.78× faster than SuiteSparse;
- never slower, and up to 7.06× faster than Eigen.

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

Performance

Upcoming **nonblocking** CG solve speedup:

- two-socket Intel Cascade Lake, 44 cores, no hyperthreads

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

- at most 0.25× slowdown, up to 2.43× faster than blocking ALP;
- at most 0.49× slowdown, up to 8.78× faster than SuiteSparse;
- never slower, and up to 7.06× faster than Eigen.

Similar results for PageRank and sparse deep neural network inference.

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

Performance

Upcoming **nonblocking** CG solve speedup:

- two-socket Intel Cascade Lake, 44 cores, no hyperthreads

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

- at most 0.25× slowdown, up to 2.43× faster than blocking ALP;
- at most 0.49× slowdown, up to 8.78× faster than SuiteSparse;
- never slower, and up to 7.06× faster than Eigen.

Similar results for PageRank and sparse deep neural network inference.

- Challenge: how to expose deeper pipelines?

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

Performance

Graph algorithm performance using the **hybrid auto-parallelisation**

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

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	537
4-hop BFS	48.8	110	54.8	99.6	83.0	74.2	23.3
20-hop 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.

Performance

Graph algorithm performance using the **hybrid auto-parallelisation**

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

	Ivy Bridge nodes						
	4	5	6	7	8	9	10
Input	1524	1271	1067	943	691	662	537
4-hop BFS	48.8	110	54.8	99.6	83.0	74.2	23.3
20-hop 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.

Interoperability with Spark, without re-writing software (2019):

				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`.
- Spark Clueweb PR: out of memory. Literature: Blogel, 128 nodes!

Ref.: Lightweight Parallel Foundations: a model-compliant communication layer by W. J. Suijlen and Y., `https://arxiv.org/abs/1906.03196` (2019)

Algebraic Programming

How far can we take this type of programming?

ALP/Dense

Classical **dense** linear algebra algorithms:

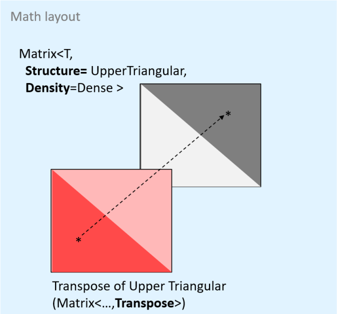
- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

ALP/Dense

Classical **dense** linear algebra algorithms:

- submatrix selection, row permutations, … random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP:



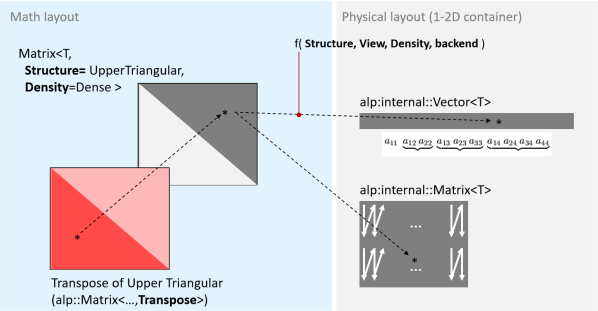
ALP/Dense

Classical **dense** linear algebra algorithms:

- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP.

- ALP is free to choose the underlying data layout:



ALP/Dense

Classical **dense** linear algebra algorithms:

- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP.

Multi-stage compilation at run-time:

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

ALP/Dense

Classical **dense** linear algebra algorithms:

- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP.

Multi-stage compilation at run-time:

1) **lazy-evaluate** ALP primitives (alike to nonblocking!)
2) when pipelines execute, instead **translate to MLIR**;

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

ALP/Dense

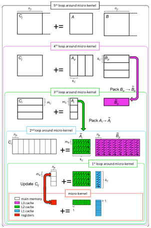
Classical **dense** linear algebra algorithms:

- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP.

Multi-stage compilation at run-time:

1) **lazy-evaluate** ALP primitives (alike to nonblocking!)
2) when pipelines execute, instead **translate to MLIR**;
3) **BLIS-like approach** to optimise generated MLIR modules



ALP/Dense

Classical dense linear algebra algorithms:

- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP.

Multi-stage compilation at run-time:

1) **lazy-evaluate** ALP primitives (alike to nonblocking!)
2) when pipelines execute, instead **translate to MLIR**;
3) **BLIS-like approach** to optimise generated MLIR modules
    - use offline auto-tuning, **once** per new architecture: ✓
    - by J. Ye, G. Rossini, S. Varoumas, a.o. at Huawei Cambridge

ALP/Dense

Classical dense linear algebra algorithms:

- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP.

Multi-stage compilation at run-time:

1) **lazy-evaluate** ALP primitives (alike to nonblocking!)
2) when pipelines execute, instead **translate to MLIR**;
3) **BLIS-like approach** to optimise generated MLIR modules
4) complete compilation of optimised MLIR modules

ALP/Dense

Classical dense linear algebra algorithms:

- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP.

Multi-stage compilation at run-time:

1) **lazy-evaluate** ALP primitives (alike to nonblocking!)
2) when pipelines execute, instead **translate to MLIR**;
3) **BLIS-like approach** to optimise generated MLIR modules
4) complete compilation of optimised MLIR modules
5) threads and/or processes execute compiled modules

ALP/Dense

Classical **dense** linear algebra algorithms:

- submatrix selection, row permutations, ... random sampling?
- ongoing work with D. G. Spampinato, V. Dimić, D. Jelovina, Y.

Introduce container structures and views to ALP.

Multi-stage compilation at run-time:

1) **lazy-evaluate** ALP primitives (alike to nonblocking!)
2) when pipelines execute, instead **translate to MLIR**;
3) **BLIS-like approach** to optimise generated MLIR modules
4) complete compilation of optimised MLIR modules
5) threads and/or processes execute compiled modules

Amounts to **delayed compilation** of 'universal binaries', with

- high-level MLIR dialects as an **architecture-agnostic** language.
- Capture algebraic information and optimisations in compiler logic.

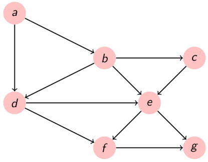
Ref.: MOM: Matrix Operations in MLIR by L. Chelini, H. Barthels, P. Bientinesi, M. Copic, T. Grosser, D. G. Spampinato, in IMPACT at HiPEAC Budapest, Hungary (2022).

ALP as a more fundamental programming model

ALP/Pregel: vertex-centric programming

- arguably, like MapReduce, a more humble API:

```
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
```



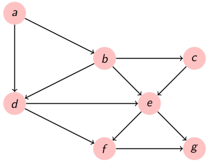
Ref.: Y., "Humble Heroes", Communications of Huawei Research, accepted. http://albert-jan.yzelman.net/publications.php#yzelman22

ALP as a more fundamental programming model

ALP/Pregel: vertex-centric programming

- arguably, like MapReduce, a more humble API:

```
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
```



Compiles using the standard ALP software stack:

- `grbcxx -b hybrid myPregelAlgo pregelAlgo.cpp`
- in 'develop' branch on GitHub, slated for v0.7 release

Ref.: Y., "Humble Heroes", Communications of Huawei Research, accepted. http://albert-jan.yzelman.net/publications.php#yzelman22

HyperDAGs

As another novel backend example, the following two steps:

1) `grbcxx -b hyperdags myProgram example.cpp`
2) `grbrun -b hyperdags ./myProgram`

dumps a **HyperDAG** corresponding to the computation.

HyperDAGs generalise DAGs: each net corresponds to one output, possibly consumed by one or more subsequent computations.

Papp, Anegg, and Y. show, amongst others, that:

- HyperDAG partitioning remains NP-Complete,
- $\nexists$ algorithm in P with reasonable approximation factors[1], and
- hierarchical NUMA-aware partitioning can be very suboptimal.

On GitHub: HyperDAG backend (branch 274), HyperDAG database.

1: assuming that the exponential time hypothesis holds.

Ref. Pál András Papp, Georg Anegg, Y., "Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications" (submitted), ArXiV:2208.08257.

Conclusion

Solving the looming software portability and productivity crisis requires

blurring the lines between compiler, library, and application

Conclusion

Solving the looming software portability and productivity crisis requires

blurring the lines between compiler, library, and application

Requires a **humble programming paradigm** that, ideally,

- achieves **hero performance**,
- **integrates easily** into applications,
- **generalises** oft-used humble programming models, and
- has **architecture portability** through, e.g., delayed compilation.

Conclusion

Solving the looming software portability and productivity crisis requires

blurring the lines between compiler, library, and application

Requires a **humble programming paradigm** that, ideally,

- achieves **hero performance**,
- **integrates easily** into applications,
- **generalises** oft-used humble programming models, and
- has **architecture portability** through, e.g., delayed compilation.

Our take with C++ ALP and MLIR is **open source**:

- `github.com/Algebraic-Programming`, Apache 2.0;
- `algebraic-programming.github.io`.

Acknowledgements

Thank you

The **GraphBLAS community**

J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.
Kepner, Gilbert, Buluç, Mattson, Moreira, and many others: https://www.graphblas.org

Algebraic programming inspired by

Generic Programming and C++ STL principles, **Alexander Stepanov**'s work in particular.

Contributors and inspirators to ALP/GraphBLAS:

Aristeidis Mastoras, Sotiris Anagnostidis, Anders Hansson;
Lorenzo Chelini, Daniele G. Spampinato, Valdimir Dimić;
Alberto Scolari, Denis Jelovina, Verner Vlačić, Auke Booij;
Aikaterini Karanasiou, Dan Iorga, Gabriel Gjini, Pouya Pourjafar;
Daniel Di Nardo, Jonathan M. Nash, Wijnand J. Suijlen;
Pál András Papp, Georg Anegg, Bill McColl;
and other colleagues – within Huawei, and our external research partners

Open source, Apache 2.0, welcome to try, use, and work together!

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

Backup slides

Backup slides

History

Algebraic Programming:

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)

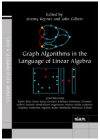
Graph Algorithms in the Language of Linear Algebra,
Jeremy Kepner & John Gilbert (2011)

From Mathematics to Generic Programming,
Alexander Stepanov & Daniel Rose (2015)

GraphBLAS.org, following work by Kepner & Gilbert,
Kepner, Gilbert, Buluç, Mattson, et alii (2016)

ALP (2022), and perhaps more?

ALP/GraphBLAS

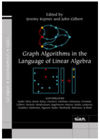


`GraphBLAS.org`; Kepner, Gilbert, Buluç, Mattson, Moreira, ...

- 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, ...

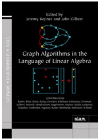
- for example, $y = A^k x$, parametrised in a semiring:

```
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 );
        }
}
```

Solves different problems for 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 );
        }
}
```

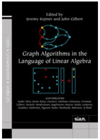
Solves different problems for different semirings:

- 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:

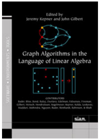



```
template< typename Semiring, typename NonzeroT, typename VectorT >
grb::RC mpv(
                grb::Vector< VectorT >  &y,
        const   grb::Matrix< NonzeroT > &A, const size_t k,
        const   grb::Vector< VectorT >  &x,
                grb::Vector< VectorT >  &buffer,
        const   Semiring                &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 );
        }
}
```

Solves different problems for different semirings:

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

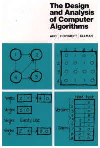
ALP/GraphBLAS



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 );
        }
}
```



Solves different problems for 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

The ALP/GraphBLAS backends in more detail

- **reference**; sparse accumulators for vectors, Gustavson's matrix format, direction optimisation, and auto-vectorisation;
- **shared-memory parallel**: OpenMP and NUMA-aware;
- **distributed-memory parallel**: LPF (MPI or ibverbs)
  - (1D) block-cyclic vector and matrix distribution;
  - sparsity-aware allgather and reduce-scatter;
  - parallel I/O.
- **hybrid**: composes distributed- with shared-memory backend.

Non-blocking ALP/GraphBLAS

Nonblocking backend. Suppose $s = r + \alpha v$ over a given `semiring`:

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

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

Blocking execution: the vector s is accessed *twice*

Non-blocking ALP/GraphBLAS

Nonblocking backend. Suppose $s = r + \alpha v$ over a given `semiring`:

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

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

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

Non-blocking ALP/GraphBLAS

Nonblocking backend. Suppose $s = r + \alpha v$ over a given `semiring`:

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

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

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

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

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

Non-blocking ALP/GraphBLAS

Nonblocking backend. Suppose $s = r + \alpha v$ over a given semiring:

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

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

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

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

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

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

Non-blocking ALP/GraphBLAS

Nonblocking backend. Suppose $s = r + \alpha v$ over a given semiring:

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

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

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

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

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

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

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).

Non-blocking ALP/GraphBLAS

Nonblocking backend. Suppose $s = r + \alpha v$ over a given semiring:

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

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

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

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

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

- *lazily* evaluate ALP/GraphBLAS calls, build pipelines

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).

Non-blocking ALP/GraphBLAS

Nonblocking backend. Suppose $s = r + \alpha v$ over a given `semiring`:

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

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

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

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

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

- *lazily* evaluate ALP/GraphBLAS calls, build pipelines

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).

Non-blocking ALP/GraphBLAS

Nonblocking backend. Suppose $s = r + \alpha v$ over a given semiring:

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

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

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

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

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

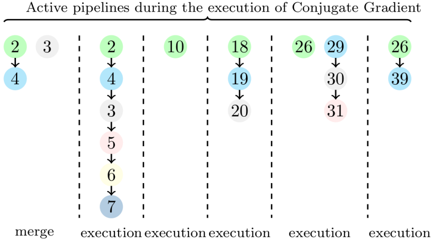
Automatic non-blocking execution (Mastoras et al., '22): humble ✓

- *lazily* evaluate ALP/GraphBLAS calls, build pipelines
- triggered when needed, **fusion without ALP program changes**

Ref.: Nonblocking execution in GraphBLAS by Aristeidis Mastoras, Sotiris Anagnostidis, and Y. in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).

Non-blocking Conjugate Gradients in ALP/GraphBLAS

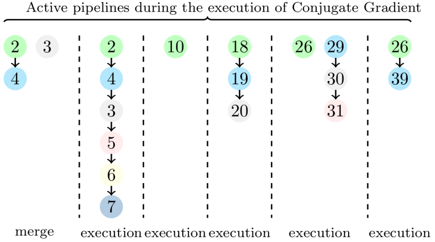
Dynamic **on-line** dependence analysis:




```
// six-stage pipeline, vectors(temp, r, x, b, u)
grb::set(temp, 0);
grb::set(r, 0);
grb::mxv(temp, A, x, ring);
grb::eWiseApply(r, b, temp, minus);
grb::set(u, r);
grb::dot(sigma, r, r, ring);

// single-stage pipeline, vector(b)
grb::dot(bnorm, b, b, ring);

tol *= sqrt(bnorm);

iter = 0;

do {
    // three-stage pipeline, vectors(temp, u)
    grb::set(temp, 0);
    grb::mxv(temp, A, u, ring);
    grb::dot(residual, temp, u, ring);

    grb::apply(alpha, sigma, residual, divide);

    // part of a two-stage pipeline, vectors (x, u, r)
    // the eWiseMulAdd at the bottom is the second stage
    grb::eWiseMulAdd(x, alpha, u, x, ring);

    // three-stage pipeline, vectors(temp, r)
    grb::eWiseMul(temp, alpha, temp, ring);
    grb::eWiseApply(r, r, temp, minus);
    grb::dot(residual, r, r, ring);

    if (sqrt(residual) < tol) break;

    grb::apply(alpha, residual, sigma, divide);

    // part of a two-stage pipeline, vectors (x, u, r)
    // the eWiseMulAdd aboce is the first stage
    grb::eWiseMulAdd(u, alpha, u, r, ring);

    sigma = residual;
} while (++iter < max_iterations);
```

Non-blocking Conjugate Gradients in ALP/GraphBLAS

Dynamic **on-line** dependence analysis:



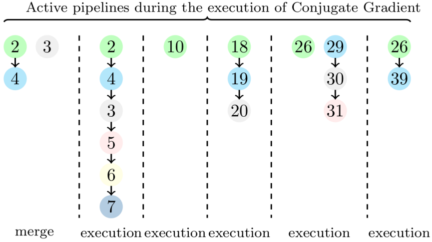
Fused execution can **cross control flow**:

- 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);
```

Non-blocking Conjugate Gradients in ALP/GraphBLAS

Dynamic **on-line** dependence analysis:



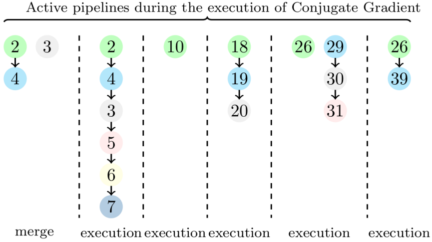
Fused execution can **cross control flow**:

- 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);
```

Non-blocking Conjugate Gradients in ALP/GraphBLAS

Dynamic **on-line** dependence analysis:



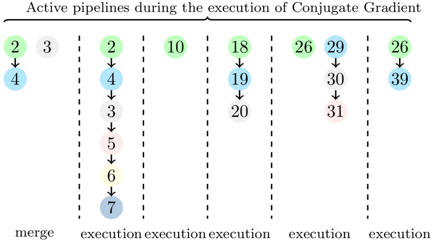
Fused execution can **cross control flow**:

- 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);
```

Non-blocking Conjugate Gradients in ALP/GraphBLAS

Dynamic **on-line** dependence analysis:




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

Fused execution can **cross control flow**:

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