Sets in the Language of Linear Algebra

Applications and Acceleration

Albert-Jan N. Yzelman

Computing Systems Laboratory, Zürich Research Center, Switzerland

3rd of March, 2026

Tile-based programming

> **Tile-based programming** is a development paradigm that abstracts individual thread management into operations on discrete, local data blocks called **tiles**, allowing the compiler and run-time to automate hardware-specific optimisations like memory movement and **Tensor Core** utilisation.
>
> – Gemini, March '26

Increasingly popular:

- Triton;
- CuTile;
- Tilelang;
- **PyPTO**

Tile-based programming

> **Tile-based programming** is a development paradigm that abstracts individual thread management into operations on discrete, local data blocks called **tiles**, allowing the compiler and run-time to automate hardware-specific optimisations like memory movement and **Tensor Core** utilisation.
>
> – Gemini, March '26

Increasingly popular:

- Triton;
- CuTile;
- Tilelang;
- **PyPTO**: **P**arallel **T**ile **O**ptimisation framework.

Tile-based programming

Not all tile-based frameworks are the same

Tile-based programming

Not all tile-based frameworks are the same–e.g., matrix addition:

- Triton (run using an $m/BI \times n/BJ$ process grid):

```
s = tl.program_id( 0 ); t = tl.program_id( 1 );
off_i = s * BI + tl.arange( 0, BI );
off_j = t * BJ + tl.arange( 0, BJ );
a_tile = tl.load( A_ptr + off_i * n + off_j );
b_tile = tl.load( B_ptr + off_i * n + off_j );
c_tile = a_tile + b_tile;
tl.store( C_ptr + off_i * n + off_j, c_tile );
```

Tile-based programming

Not all tile-based frameworks are the same–e.g., matrix addition:

- Triton (run using an $m/BI \times n/BJ$ process grid):

```
s = tl.program_id( 0 ); t = tl.program_id( 1 );
off_i = s * BI + tl.arange( 0, BI );
off_j = t * BJ + tl.arange( 0, BJ );
a_tile = tl.load( A_ptr + off_i * n + off_j );
b_tile = tl.load( B_ptr + off_i * n + off_j );
c_tile = a_tile + b_tile;
tl.store( C_ptr + off_i * n + off_j, c_tile );
```

- PyPTO (call using standard Torch tensors and JIT):

```
# A, B, C: pypto.Tensor( shape, pypto.DT_FP32 );
pypto.set_vec_tile_shapes( 1, 4 );
C = A + B;
```

Tile-based programming

Main difference:

- Triton and like: user decides memory layout, tiling, blocking, and writes tiled algorithm **explicitly**;
- "Tensor-level" PyPTO: users express computation, but PTO decides data layout and produces a tiled algorithm **implicitly**.

Tile-based programming

Main difference:

- Triton and like: user decides memory layout, tiling, blocking, and writes tiled algorithm **explicitly**;
- "Tensor-level" PyPTO: users express computation, but PTO decides data layout and produces a tiled algorithm **implicitly**.

Control via a **separation of concerns**, functional v. performance:

- e.g., the setting of the tile size for vector or cube computations.

Tile-based programming

Main difference:

- Triton and like: user decides memory layout, tiling, blocking, and writes tiled algorithm **explicitly**;
- "Tensor-level" PyPTO: users express computation, but PTO decides data layout and produces a tiled algorithm **implicitly**.

Control via a **separation of concerns**, functional v. performance:

- e.g., the setting of the tile size for vector or cube computations.

Nevertheless, this approach pushes a **hard problem** to the framework:

- the tensor-level problem is decomposed into a fine-grained graph;
- which it schedules over multiple vector and cube cores; and
- may need to distribute over multiple dies, devices, nodes.

Tile-based programming

Offline general DAG scheduling.

Tile-based programming

Offline general DAG scheduling. Assume **realistic costs**:

- compute, data movement (g), *and* latency (l)– i.e., BSP.

Assume work must be near-balanced (up to some $\epsilon > 0$).

Tile-based programming

Offline general DAG scheduling. Assume **realistic costs**:

- compute, data movement (g), *and* latency (l)– i.e., BSP.

Assume work must be near-balanced (up to some $\epsilon > 0$). Then:

Theorem

BSP scheduling is NP-hard, already for DAGs of height 2 and in-trees.

Tile-based programming

Offline general DAG scheduling. Assume **realistic costs**:

- compute, data movement (g), *and* latency (l)– i.e., BSP.

Assume work must be near-balanced (up to some $\epsilon > 0$). Then:

Theorem

BSP scheduling is NP-hard, already for DAGs of height 2 and in-trees.

Consider the **parallel overhead** $O(p) = T_{\text{opt}}(p) - T_{\text{seq}}/p$. Then:

Theorem

*Assuming the exponential time hypothesis, $p = 2$, and some $\delta > 0$, O is **NP-hard to approximate** to an $n^{1/(\log \log n)^{\delta}}$ factor.*

This holds with or without **replication**.

Can still do well in practice: **OneStopParallel** scheduling toolkit.

Ref.: Replication in Graph Partitioning and Scheduling Problems, Papp, Böhnlein, Y. (under preparation)
Ref.: DAG Scheduling in the BSP Model, Papp, Anegg, Y., SOFSEM '25, **best paper**.
Ref.: `github.com/Algebraic-Programming/OneStopParallel` (Apache 2.0).

Tile-based programming

PyPTO:

- provides high-level C++ and Python programming for accelerators;
- separates functional specification from performance parameters;
- reduces, in the general case, to fine-grained DAG scheduling;
- applies, at current, to contemporary AI workloads (tensors);
- is open-source: `gitcode.com/cann/pypto`.

Related research questions:

- how can we make tile-based programming more **general-purpose**?
- can we **avoid DAG scheduling** in large subclasses of problems?

Tile-based programming

PyPTO:

- provides high-level C++ and Python programming for accelerators;
- separates functional specification from performance parameters;
- reduces, in the general case, to fine-grained DAG scheduling;
- applies, at current, to contemporary AI workloads (tensors);
- is open-source: `gitcode.com/cann/pypto`.

Related research questions:

- how can we make tile-based programming more **general-purpose**?
- can we **avoid DAG scheduling** in large subclasses of problems?

Algebraic Programming addressed similar questions for linear algebra;

- we consider **unordered sets** in both ALP and tile-based contexts.

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

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:

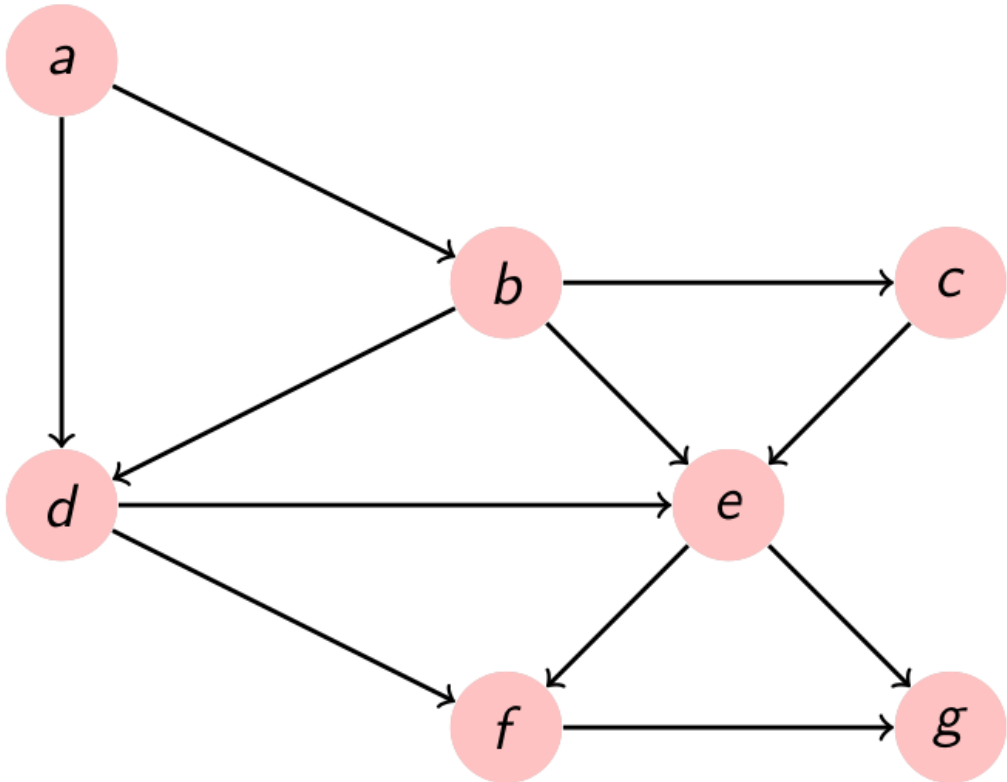
- linear algebra: **ALP/GraphBLAS** and **ALP/Dense**,
- vertex-centric programming: **ALP/Pregel**,
- towards tensor algebra: **ALP/Tensors** and **ALP/Ascend**,
- ...

Interoperability with existing software:

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

ALP/Pregel

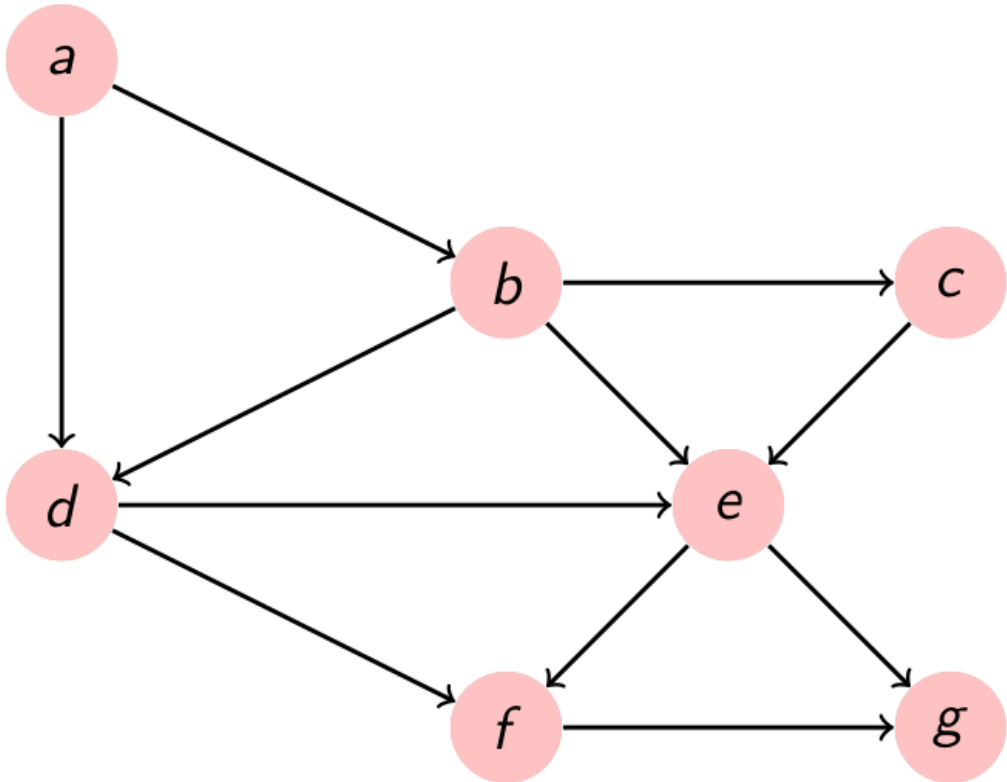
Pregel:



- 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

Example application: PageRank with vertex inactivation.

- initialise **active vertex set** $A = V$;
- initialise equally-distributed local score: $\pi_i = 1/n$.
- Then, round-by-round, for each vertex $i \in A$:
  - divide π_i by the number of neighbours and broadcast it;
  - let x_i be the sum of incoming messages from neighbours in A;
  - set new $\tilde{\pi}_i = \alpha + (1 - \alpha)x_i$;
  - **if** $|\tilde{\pi}_i - \pi_i| < tol$ **then**
    remove i from active vertex set ($A = A \setminus \{i\}$);
  - $\pi_i = \tilde{\pi}_i$;
- terminate when $A = \emptyset$ (or some max_iters is reached).

While "PageRank-like", **not mathematically equivalent** to Brin & Page(!)

Ref.: Humble Heroes, Y., Communications of Huawei Research (6), pp. 146–170 (2024).

Set operations in sparse linear algebra

Can represent A linear algebraically as a **sparse vector** a:

- union: element-wise OR, intersection: element-wise AND, ...

Our "wish list":

1) insertion, iteration, and removal–ideally in $\mathcal{O}(1)$ and in parallel(!);
2) we require our data structures to be **tileable**.

Standard data structures to represent sets are

- Red-Black trees (ordered), and
- hash-maps (unordered).

Set operations in sparse linear algebra

Can represent A linear algebraically as a **sparse vector** a:

- union: element-wise OR, intersection: element-wise AND, ...

Our "wish list":

1) insertion, iteration, and removal–ideally in $\mathcal{O}(1)$ and in parallel(!);
2) we require our data structures to be **tileable**.

Standard data structures to represent sets are

- Red-Black trees (ordered), and
- hash-maps (unordered).

Our application is special:

- the number of unique vertices is limited; thus,
- we may also represent A by **sparse accumulators**.

Sparse vectors

Sparse vectors in ALP/GraphBLAS:

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

Ref.: Gilbert, Moler, and Schreiber, *Sparse matrices in MATLAB: Design and implementation.* JMAA (1992);
Ref.: Y., Di Nardo, Nash, Suijlen, *A C++ GraphBLAS* (2020).

Sparse vectors

Sparse vectors in ALP/GraphBLAS:

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

Ref.: Gilbert, Moler, and Schreiber, *Sparse matrices in MATLAB: Design and implementation.* JMAA (1992);
Ref.: Y., Di Nardo, Nash, Suijlen, *A C++ GraphBLAS* (2020).

Alternatively, a tree-based map (`std::map`):

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

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

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

Set operations in sparse linear algebra

Every sparse vector in shared-memory parallel ALP/GraphBLAS has:

1) a global SPA of size n, which, in parallel regions, is read-only;
2) $\lceil n/t \rceil$ tile-local SPAs of size $t \ll n$, where t is the *tile size* and
   - the value- and bit-arrays are pointers into the global SPA;
   - the stacks are tile-local instead: $\Theta(n)$ memory overhead.

Set operations in sparse linear algebra

Every sparse vector in shared-memory parallel ALP/GraphBLAS has:

1) a global SPA of size n, which, in parallel regions, is read-only;
2) $\lceil n/t \rceil$ tile-local SPAs of size $t \ll n$, where t is the *tile size* and
   - the value- and bit-arrays are pointers into the global SPA;
   - the stacks are tile-local instead: $\Theta(n)$ memory overhead.

Each thread dynamically processes different tiles. For each tile:

1) the thread initialises its local SPA; $\Theta(n)$ *total* overhead.
2) process it, flag modifications to global bit-array; $\Theta(n/t)$ overhead.

Multiple operations may execute on a tile if data dependences allow.

Set operations in sparse linear algebra

Every sparse vector in shared-memory parallel ALP/GraphBLAS has:

1) a global SPA of size n, which, in parallel regions, is read-only;
2) $\lceil n/t \rceil$ tile-local SPAs of size $t \ll n$, where t is the *tile size* and
   - the value- and bit-arrays are pointers into the global SPA;
   - the stacks are tile-local instead: $\Theta(n)$ memory overhead.

Each thread dynamically processes different tiles. For each tile:

1) the thread initialises its local SPA; $\Theta(n)$ *total* overhead.
2) process it, flag modifications to global bit-array; $\Theta(n/t)$ overhead.

Multiple operations may execute on a tile if data dependences allow.

Finally, after all tiles and operations thereon are done:

3) for each modified tile i, prefix-sum their k_i new nonzeroes and copy them into the global stack; $\Theta(n/t + Tg + l + \sum k_i)$ overhead.

Set operations in sparse linear algebra

Notable alternative designs:

1) a global SPA of size n, which, in parallel regions, is read-only;
2) (up to) T tile-local SPAs of size $t \ll n$, where t is the *tile size*.
   - The stacks are now thread-local: $\Theta(Tt)$ memory overhead;
   - requires that $t \leq n/T$.

Each thread dynamically processes different tiles. For each tile:

1) the thread initialises its local SPA; $\Theta(n)$ *total* overhead.
2) process it, flag modifications to global bit-array; $\Theta(n/t)$ overhead.
   - as before, multiple operations on the same tile may execute

Set operations in sparse linear algebra

Notable alternative designs:

1) a global SPA of size n, which, in parallel regions, is read-only;
2) (up to) T tile-local SPAs of size $t \ll n$, where t is the *tile size*.
   - The stacks are now thread-local: $\Theta(Tt)$ memory overhead;
   - requires that $t \leq n/T$.

Each thread dynamically processes different tiles. For each tile:

1) the thread initialises its local SPA; $\Theta(n)$ *total* overhead.
2) process it, flag modifications to global bit-array; $\Theta(n/t)$ overhead.
   - as before, multiple operations on the same tile may execute
3) after processing T tiles concurrently, prefix-sum new nonzeroes and copy them into the global stack; $\Theta(n/t(g+l)+k)$ overhead.

Set operations in sparse linear algebra

Notable alternative designs:

1) a global SPA of size n, which, in parallel regions, is read-only;
2) (up to) T tile-local SPAs of size $t \ll n$, where t is the *tile size*.
   - The stacks are now thread-local: $\Theta(Tt)$ memory overhead;
   - requires that $t \leq n/T$.
3) one more global SPA of size n/t (read-only); $\Theta(n/t)$ overhead.

Each thread dynamically processes different non-empty tiles:

1) the thread initialises its local SPA; $\mathcal{O}(\max\{nz, n\}$ overhead.
2) process it, flag modifications; $\mathcal{O}(\max\{nz + k, n/t\}$ overhead.
   - as before, multiple operations on the same tile may execute
3) after processing T tiles concurrently, prefix-sum new nonzeroes and copy them; $\mathcal{O}(\max\{nz + k, n/t\}(g + l) + k)$ overhead.

Set operations in sparse linear algebra

Notable alternative designs:

1) a global SPA of size n, which, in parallel regions, is read-only;
2) (up to) T tile-local SPAs of size $t \ll n$, where t is the *tile size*.
   - The stacks are now thread-local: $\Theta(Tt)$ memory overhead;
   - requires that $t \leq n/T$.
3) one more global SPA of size n/t (read-only); $\Theta(n/t)$ overhead.

Each thread dynamically processes different non-empty tiles:

1) the thread initialises its local SPA; $\mathcal{O}(\max\{nz, n\}$ overhead.
2) process it, flag modifications; $\mathcal{O}(\max\{nz + k, n/t\}$ overhead.
   - as before, multiple operations on the same tile may execute
3) after processing T tiles concurrently, prefix-sum new nonzeroes and copy them; $\mathcal{O}(\max\{nz + k, n/t\}(g + l) + k)$ overhead.

These alternatives indeed were outperformed by the basic variant.

Results

We evaluate on a dual-socket Intel Xeon Gold 6238T, 44 cores;

- PageRank with vertex inactivation, with and without tiling;
- GCC 13.3.0, Ubuntu 24.04.2 LTS, ALP/GraphBLAS pre-v0.8;
- t is selected s.t. work set fits private cache, T s.t. n fits in cache:

Matrix	size	nonzeroes	serial	no tiling	**tiled**
gyro_m	17'361	340'431	0.63 ms	**1.06**×	0.60×
vanbody	47'072	2'329'056	2.87 ms	3.74×	**5.82**×
...					

Results

We evaluate on a dual-socket Intel Xeon Gold 6238T, 44 cores;

- PageRank with vertex inactivation, with and without tiling;
- GCC 13.3.0, Ubuntu 24.04.2 LTS, ALP/GraphBLAS pre-v0.8;
- t is selected s.t. work set fits private cache, T s.t. n fits in cache:

Matrix	size	nonzeroes	serial	no tiling	**tiled**
gyro_m	17'361	340'431	0.63 ms	**1.06**×	0.60×
vanbody	47'072	2'329'056	2.87 ms	3.74×	**5.82**×
inline_1	503'712	36'816'170	44.41 ms	10.54×	**16.94**×
parabolic_fem	525'825	3'674'625	1.07 ms	**1.28**×	0.24×
...					

Results

We evaluate on a dual-socket Intel Xeon Gold 6238T, 44 cores;

- PageRank with vertex inactivation, with and without tiling;
- GCC 13.3.0, Ubuntu 24.04.2 LTS, ALP/GraphBLAS pre-v0.8;
- t is selected s.t. work set fits private cache, T s.t. n fits in cache:

Matrix	size	nonzeroes	serial	no tiling	**tiled**
gyro_m	17'361	340'431	0.63 ms	**1.06**×	0.60×
vanbody	47'072	2'329'056	2.87 ms	3.74×	**5.82**×
inline_1	503'712	36'816'170	44.41 ms	10.54×	**16.94**×
parabolic_fem	525'825	3'674'625	1.07 ms	**1.28**×	0.24×
Serena	1'391'349	64'131'971	75.52 ms	8.24×	8.46×
...					

Results

We evaluate on a dual-socket Intel Xeon Gold 6238T, 44 cores;

- PageRank with vertex inactivation, with and without tiling;
- GCC 13.3.0, Ubuntu 24.04.2 LTS, ALP/GraphBLAS pre-v0.8;
- t is selected s.t. work set fits private cache, T s.t. n fits in cache:

Matrix	size	nonzeroes	serial	no tiling	**tiled**
gyro_m	17'361	340'431	0.63 ms	**1.06**×	0.60×
vanbody	47'072	2'329'056	2.87 ms	3.74×	**5.82**×
inline_1	503'712	36'816'170	44.41 ms	10.54×	**16.94**×
parabolic_fem	525'825	3'674'625	1.07 ms	**1.28**×	0.24×
Serena	1'391'349	64'131'971	75.52 ms	8.24×	8.46×
...					
adaptive	6'815'744	27'248'640	15.90 ms	**3.15**×	1.02×
uk-2002	18'520'486	298'113'762	432.78 ms	8.48×	8.74×
road_usa	23'947'347	57'708'624	786 ms	6.21×	**10.41**×

Recall: non-tiled solution is $\Theta(nz + k)$, whereas tiled version is $\Theta(n)$.

Ref.: Efficient handling of sparse vectors for parallel nonblocking execution in GraphBLAS, Mastoras & Y., Euro-Par 2025 Workshops (to appear)

Conclusion and Outlook

We explored a **tiled data structure** for sets with limited domains. It:

- exploits a bijection between set and sparse vector operations;
- investigates how to parallelise and tile operations on SPAs;
- identify primarily **synchronisation-memory trade-offs**.

We placed these in context of general-purpose tile-based programming

- while avoiding solving inapproximable scheduling problems; and
- in the context of Algebraic Programming and GraphBLAS.

However, **results are less convincing** v. tiling dense vectors (2.43×).

In future,

- analytic models to drive trade-offs for better overall performance;
- **tiled tree** (and/or tiled hash tables) should be investigated.

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:

- Y., Di Nardo, Nash, Suijlen: A C++ GraphBLAS (2020);
- Mastoras, Anagnostidis, Y.: Design and implementation for nonblocking execution in GraphBLAS: tradeoffs and performance, ACM TACO (2023);
- Scolari, Y.: Effective implementation of the High Performance Conjugate Gradient benchmark on ALP/GraphBLAS, IPDPSW (GrAPL 2023);
- Spampinato, Jelovina, Zhuang, Y.: Towards Structured Algebraic Programming, ACM ARRAY (2023);
- Papp, Anegg, Y.: Partitioning Hypergraphs is Hard: Models, Inapproximability, and Applications, ACM SPAA (2023);
- Papp, Anegg, Karanasiou, Y.: Efficient Multi-Processor Scheduling in Increasingly Realistic Models, ACM SPAA (2024);
- Y.: Humble Heroes, Communications of Huawei Research (2024);
- Pasadakis, Schenk, Vlaçić, Y.: Nonlinear spectral clustering with C++ GraphBLAS, extended abstract, IEEE HPEC (2023, **outstanding short paper**);
- Papp, Anegg, Y.: DAG scheduling in the BSP model, SOFSEM (2025, **best paper**);
- Niu, Meyer, Pasadakis, Y., Schenk: Incremental Sparse Tensor Format for Maximizing Efficiency in Tensor–Vector Multiplications, IEEE Cluster (2025, **best poster**);
- Martinez-Ferrer, Y., Beltran: Distributed and heterogeneous tensor-vector contraction algorithms for high performance computing, Elsevier FGCS (2025);
- Papp, Sobczyk, Y.: The Impact of Partial Computations on the Red-Blue Pebble Game, ACM SPAA (2025);
- Papp, Böhnlein, Y.: Replication in Graph Partitioning and Scheduling Problems (in submission);
- Mastoras, Y.: Efficient handling of sparse vectors for parallel nonblocking execution in GraphBLAS, GraphSys at EuroPar (2025, to appear).

Backup slides

Backup slides

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Performance

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

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

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

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

Performance

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

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

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

The nonblocking backend:

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

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

Performance

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

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

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

The nonblocking backend:

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

Similar results for PageRank and sparse deep neural network inference.

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

Tile-based programming

In case of a streaming application (e.g., training neural networks),

- the problem reverts to balanced hypergraph partitioning:

Theorem

Assuming the exponential time hypothesis, it is NP-hard to approximate the balanced hypergraph partitioning problem to an $n^{1/(\log\log n)^{\delta}}$ *factor.*

When allowing replication:

Theorem

For any $0 < \epsilon < 1$*, graph partitioning with replication is NP-hard to approximate to any finite factor as well as to an* $n^{(2-\delta)}$ *additive term.*

(This theorem also applies to hypergraph partitioning.)

Ref.: Partitioning Hypergraphs is Hard, Papp, Anegg, Y., SPAA '23
Ref.: The Impact of Partial Computations on the Red-Blue Pebble Game, Papp, Sobczyk, Y, SPAA '25
Ref.: Replication in Graph Partitioning and Scheduling Problems, Papp, Böhnlein, Y. (under preparation)

Practical scheduling in realistic models

We show three types of **experimental results**:

1) **model-based** evaluation– i.e., (Multi-)BSP cost;
2) **real-world** evaluation on sparse triangular solves– i.e., time;
3) summary of gains for **PyPTO**– i.e., speedups.

Practical scheduling in realistic models

We show three types of **experimental results**:

1) **model-based** evaluation– i.e., (Multi-)BSP cost;
2) **real-world** evaluation on sparse triangular solves– i.e., time;
3) summary of gains for **PyPTO**– i.e., speedups.

Algorithms evaluated:

- optimal schedules via **ILP formulations** (COPT);
- ILP formulations for iterative refinement
  - two supersteps, comm. optimisation;
- heuristic **initialisers**: greedy BSP (g_k, l_k), growLocal (l), Sarkar;
- **local search**: hill climbing (g_k, l_k), Kernighan-Lin (g_k, l_k);
- large-scale: **multi-level**, **divide-and-conquer** (composes former).

Practical scheduling in realistic models

We show three types of **experimental results**:

1) **model-based** evaluation– i.e., (Multi-)BSP cost;
2) **real-world** evaluation on sparse triangular solves– i.e., time;
3) summary of gains for **PyPTO**– i.e., speedups.

Algorithms evaluated:

- optimal schedules via **ILP formulations** (COPT);
- ILP formulations for iterative refinement
  - two supersteps, comm. optimisation;
- heuristic **initialisers**: greedy BSP (g_k, l_k), growLocal (l), Sarkar;
- **local search**: hill climbing (g_k, l_k), Kernighan-Lin (g_k, l_k);
- large-scale: **multi-level**, **divide-and-conquer** (composes former).

All available within a single toolbox: **OneStopParallel (OSP)**

- `github.com/Algebraic-Programming/OneStopParallel` (Apache);

Practical scheduling in realistic models

We show three types of **experimental results**:

1) **model-based** evaluation– i.e., (Multi-)BSP cost;
2) **real-world** evaluation on sparse triangular solves– i.e., time;
3) summary of gains for **PyPTO**– i.e., speedups.

Algorithms evaluated:

- optimal schedules via **ILP formulations** (COPT);
- ILP formulations for iterative refinement
  - two supersteps, comm. optimisation;
- heuristic **initialisers**: greedy BSP (g_k, l_k), growLocal (l), Sarkar;
- **local search**: hill climbing (g_k, l_k), Kernighan-Lin (g_k, l_k);
- large-scale: **multi-level**, **divide-and-conquer** (composes former).

All available within a single toolbox: **OneStopParallel (OSP)**

- `github.com/Algebraic-Programming/OneStopParallel` (Apache);
- also available: Ubuntu and RedHat packages, OSP-as-a-service.

Practical scheduling in realistic models

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

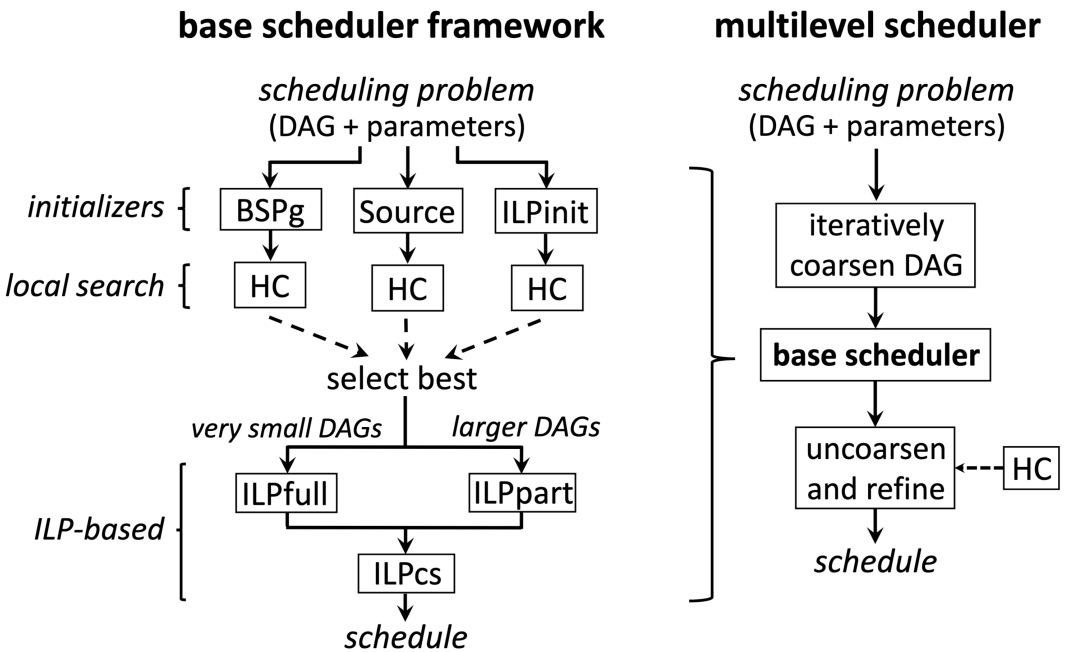
- but only for **small DAGs**: hundreds of nodes, at most.
  - CBC, improves with the use of commercial ILP solvers.
- With **memory constraints** (MPP): 80 nodes, at most (COPT).

Practical scheduling in realistic models

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

- but only for **small DAGs**: hundreds of nodes, at most.
  - CBC, improves with the use of commercial ILP solvers.
- With **memory constraints** (MPP): 80 nodes, at most (COPT).

Hence:



Ref.: Efficient Multi-Processor Scheduling in Increasingly Realistic Models, Pál András Papp, Georg Anegg, Aikaterini Karanasiou, and Y., SPAA '24.
Ref.: 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

Model-based evaluation, BSP and NUMA cost reductions.

- Diverse computational dataset from the HyperDAG database;
- HyperDAGs with 40 to 10 000 nodes, both coarse and fine-grained;
- `github.com/Algebraic-Programming/HyperDAG_DB`.

Practical scheduling in realistic models

Model-based evaluation, BSP and NUMA cost reductions.

- Diverse computational dataset from the HyperDAG database;
- HyperDAGs with 40 to 10 000 nodes, both coarse and fine-grained;
- `github.com/Algebraic-Programming/HyperDAG_DB`.

Lower costs are better, baselines ($1\times$) are Cilk and HDagg.

Practical scheduling in realistic models

Model-based evaluation, BSP and NUMA cost reductions.

- Diverse computational dataset from the HyperDAG database;
- HyperDAGs with 40 to 10 000 nodes, both coarse and fine-grained;
- `github.com/Algebraic-Programming/HyperDAG_DB`.

Lower costs are better, baselines ($1\times$) are Cilk and HDagg.

- $d = \{1, 2, 4\}$, $p = \{8, 16\}$, $g = 1$, $l = 5$, geomean:
  - uniform scheduling: $0.56\times$, $0.76\times$;
  - NUMA-aware scheduling: $0.40\times$, $0.57\times$;
  - multi-level scheduling: $0.39\times$, $0.65\times$.

Practical scheduling in realistic models

Model-based evaluation, BSP and NUMA cost reductions.

- Diverse computational dataset from the HyperDAG database;
- HyperDAGs with 40 to 10 000 nodes, both coarse and fine-grained;
- `github.com/Algebraic-Programming/HyperDAG_DB`.

Lower costs are better, baselines ($1\times$) are Cilk and HDagg.

- $d = \{1, 2, 4\}$, $p = \{8, 16\}$, $g = 1$, $l = 5$, geomean:
  - uniform scheduling: $0.56\times$, $0.76\times$;
  - NUMA-aware scheduling: $0.40\times$, $0.57\times$;
  - multi-level scheduling: $0.39\times$, $0.65\times$.
- $d = 4$, $p = 16$, $g = 1$, $l = 5$, geomean:
  - uniform scheduling: $0.57\times$, $0.70\times$;
  - NUMA-aware scheduling: $0.29\times$, $0.42\times$;
  - multi-level scheduling: $0.13\times$, $0.21\times$ (!!)
- The SPAA paper contains further experiments.

Ref.: Efficient Multi-Processor Scheduling in Increasingly Realistic Models, Pál András Papp, Georg Anegg, Aikaterini Karanasiou, and Y., SPAA '24.

Practical scheduling in realistic models

Model-based evaluation, multi-processor pebbling, ILP solvers.

- two-level with memory constraints, 40-80 node (hyper)DAGs, $p = 4$, $g = 1$, $l = 10$ (small problems), and $M = 3 \sum_{v \in V} m(v)$:
  - $0.77\times$ the cost of greedy BSP & clairvoyant LRU;
  - $0.88\times$ the cost of **optimal** BSP & clarivoyant LRU;
  - $0.66\times$ the cost of Cilk & regular LRU (**online** SotA).

Practical scheduling in realistic models

Model-based evaluation, multi-processor pebbling, ILP solvers.

- two-level with memory constraints, 40-80 node (hyper)DAGs, $p = 4$, $g = 1$, $l = 10$ (small problems), and $M = 3 \sum_{v \in V} m(v)$:
  - 0.77× the cost of greedy BSP & clairvoyant LRU;
  - 0.88× the cost of **optimal** BSP & clarivoyant LRU;
  - 0.66× the cost of Cilk & regular LRU (**online** SotA).
- larger HyperDAGs up to 264-464 nodes, $M = 5 \sum_{v} m(v)$:
  - **divide-and-conquer** ILPs: 0.66–1.13× the cost of SotA.

Practical scheduling in realistic models

Model-based evaluation, multi-processor pebbling, ILP solvers.

- two-level with memory constraints, 40-80 node (hyper)DAGs, $p = 4$, $g = 1$, $l = 10$ (small problems), and $M = 3 \sum_{v \in V} m(v)$:
  - 0.77× the cost of greedy BSP & clairvoyant LRU;
  - 0.88× the cost of **optimal** BSP & clarivoyant LRU;
  - 0.66× the cost of Cilk & regular LRU (**online** SotA).
- larger HyperDAGs up to 264-464 nodes, $M = 5 \sum_{v} m(v)$:
  - **divide-and-conquer** ILPs: 0.66–1.13× the cost of SotA.
- the above results employ **recomputation** when useful;
  - max. penalty if disallowed: 1.4× optimality.

ILP formulations solved using **commercial solvers** (COPT).

Ref.: 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, Intel CPU, 22 cores:

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, Intel CPU, 22 cores:

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(!), yet not faster;

Practical scheduling in realistic models

Sparse triangular solve by DAG scheduling, Intel CPU, 22 cores:

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(!), yet not faster;
- on AMD, 64 cores, 1.42× vs. SpMP, 2.63× vs. HDagg;
- on ARM, 48 cores, 4.29× vs. HDagg.

Practical scheduling in realistic models

Sparse triangular solve by DAG scheduling, Intel CPU, 22 cores:

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(!), yet not faster;
- on AMD, 64 cores, 1.42× vs. SpMP, 2.63× vs. HDagg;
- on ARM, 48 cores, 4.29× vs. HDagg.

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

- github.com/Algebraic-Programming/OneStopParallel

Ref.: 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).

Practical scheduling in realistic models

PyPTO – back to AI;) –

Practical scheduling in realistic models

PyPTO – back to AI;) – has additional constraints:

- modern accelerators have separate vector and **tensor units**;
- subgraphs are compiled: **isomorphism** reduces instruction size;
- replace default *Iso scheduler*, consider on-board E2E 'system tests'.

Practical scheduling in realistic models

PyPTO – back to AI;) – has additional constraints:

- modern accelerators have separate vector and **tensor units**;
- subgraphs are compiled: **isomorphism** reduces instruction size;
- replace default *Iso scheduler*, consider on-board E2E 'system tests'.

We modify Greedy BSP to fit the constraints, and reimplement Sarkar:

- in-depth evaluation: 22% geomean gains across all tests;
  - typically, the larger the operator, the higher the gains;
  - **DeepSeek v3.2 attention**: 47% speedup using Greedy BSP;

Practical scheduling in realistic models

PyPTO – back to AI;) – has additional constraints:

- modern accelerators have separate vector and **tensor units**;
- subgraphs are compiled: **isomorphism** reduces instruction size;
- replace default *Iso scheduler*, consider on-board E2E 'system tests'.

We modify Greedy BSP to fit the constraints, and reimplement Sarkar:

- in-depth evaluation: 22% geomean gains across all tests;
  - typically, the larger the operator, the higher the gains;
  - **DeepSeek v3.2 attention**: 47% speedup using Greedy BSP;
- algorithm & parameter (g, l) tuning, mod pre-scheduling passes.

Code: Böhnlein, Steiner, Matzoros, De Vita, Papp, Y., '25-'26:

- `https://gitcode.com/cann/pypto/merge_requests/905`
- `https://github.com/Algebraic-Programming/OneStopParallel`

Ref.: Partitioning parallel programs for macro-dataflow, V. Sarkar and J. Hennessy, Proc. ACM Conference on LISP and Functional Programming (1986).

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).

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 );
```

Historical context

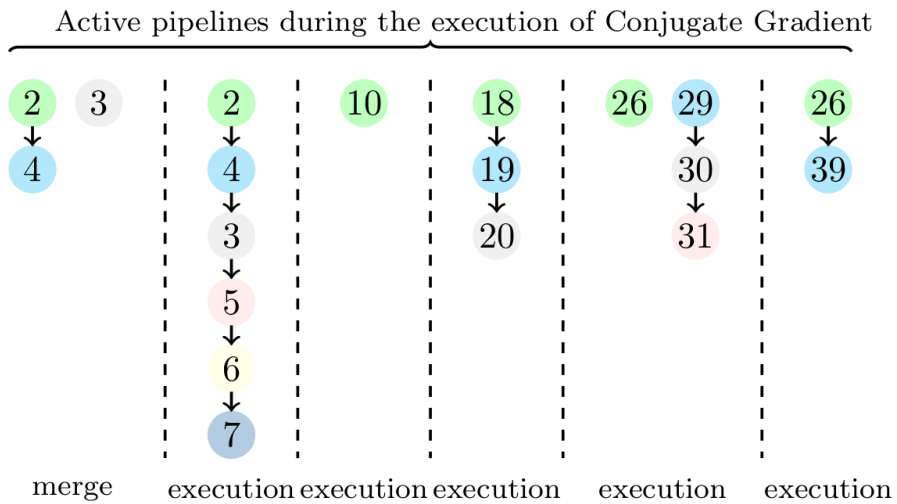
- The Design and Analysis of Computer Algorithms,
  Aho, Hopcroft, Ullman (1974)
- Introduction to Algorithms (first edition only),
  Cormen, Leiserson, Rivest (1990)
- **Elements of Programming**,
  Alexander Stepanov & Paul McJones (2009)
- **Graph Algorithms in the Language of Linear Algebra**,
  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)
- ...

The nonblocking backend

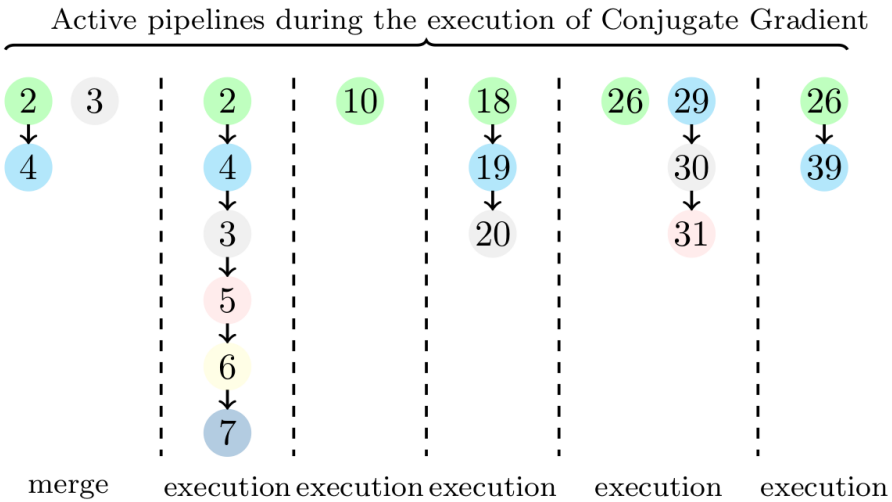
Dynamic **on-line** dependence analysis:




```
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:



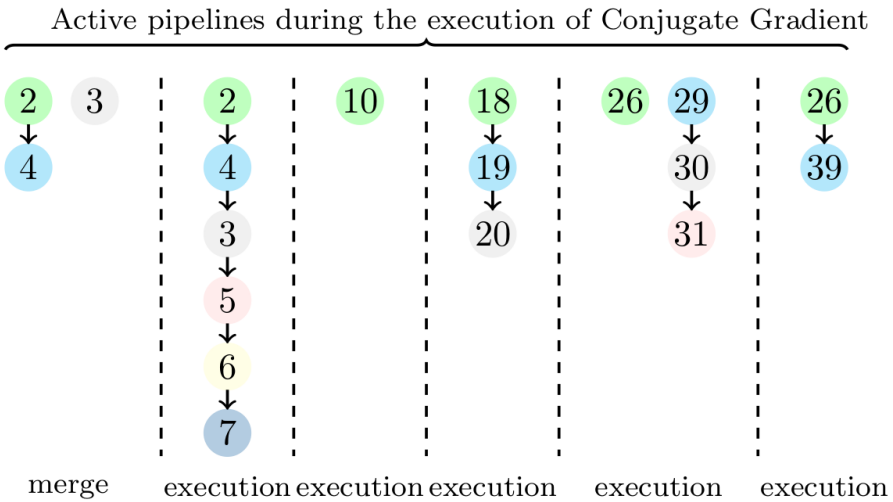
Fused execution can **cross control flow**:

- 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:



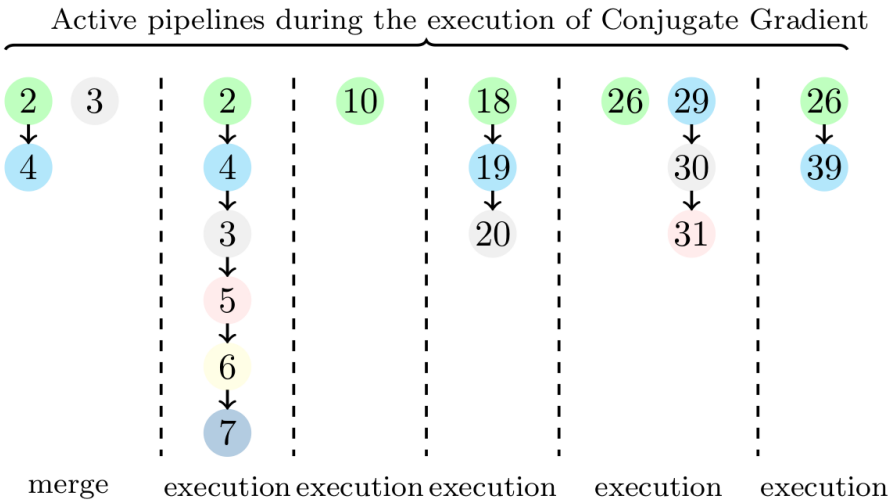
Fused execution can **cross control flow**:

- 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:



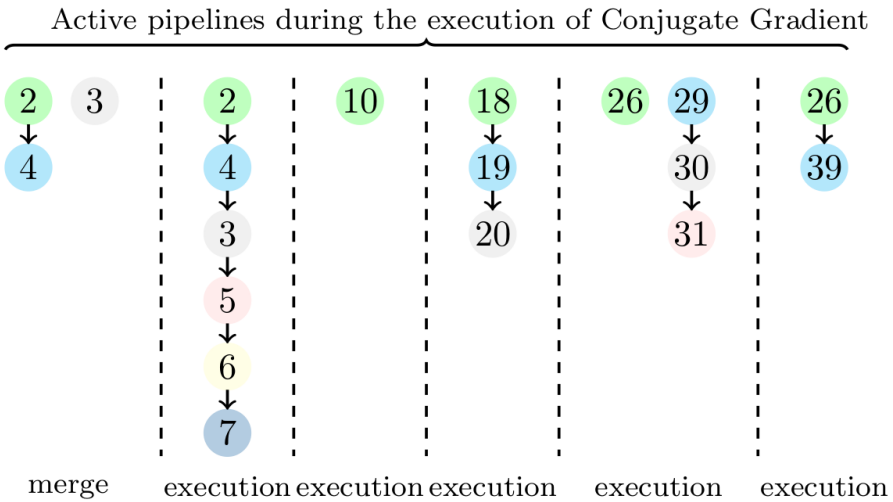
Fused execution can **cross control flow**:

- 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);
```

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...
```