(Optimal) Scheduling

in increasingly realistic models, with applications

Albert-Jan N. Yzelman

Computing Systems Laboratory, Zürich Research Center, Switzerland

HUAWEI

18th 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, 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, 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, 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;

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, 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.

Scheduling in realistic models

First, assume **realistic costs**:

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

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

Scheduling in realistic models

First, assume **realistic costs**:

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

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

2020, Jain and Zaharia: spectral partitioning to find I/O lower bounds.

Scheduling in realistic models

First, assume **realistic costs**:

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

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

2020, Jain and Zaharia: spectral partitioning to find I/O lower bounds.

2022/2023:

Theorem

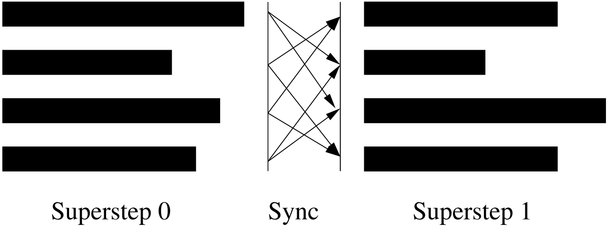
Assuming ETH, it is not possible to approximate the optimum of the ϵ-balanced hypergraph partitioning problem to an $n^{(\log\log n)^{-\delta}}$ factor in polynomial time, for some $\delta > 0$.

This already holds when the hypergraphs are DAGs.

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

Scheduling in realistic models

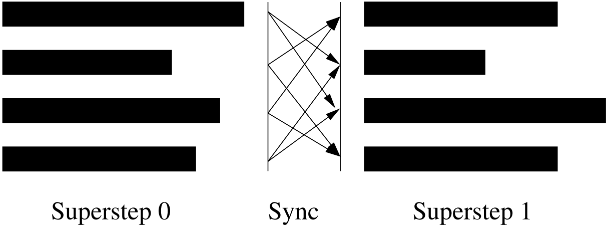
Scheduling via partitioning by imposing BSP supersteps:



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

Scheduling in realistic models

Scheduling via partitioning by imposing BSP supersteps:



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

Theorem

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

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

Scheduling in realistic models

"Pure" BSP **scheduling**, assign place *and* time to DAG vertices

Scheduling in realistic models

"Pure" BSP **scheduling**, assign place *and* time to DAG vertices:

Theorem

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

This already holds for $p = 2$.

Scheduling in realistic models

"Pure" BSP **scheduling**, assign place *and* time to DAG vertices:

Theorem

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

This already holds for $p = 2$.

When combining horizontal *and* vertical data movement:

Theorem

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

Ref.: DAG Scheduling in the BSP Model, Pál András Papp, Georg Anegg, and Y., SOFSEM '25 **best paper** award. (Theorems 2–5).
Ref.: Multiprocessor Scheduling with Memory Constraints: Fundamental Properties and Finding Optimal Solutions, Pál András Papp, Toni Böhnlein, and Y., ICPP '25. (Theorem 1).

Scheduling in realistic models

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

Scheduling in realistic models

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

Scheduling in realistic models

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

Remarks:

- $T_{\text{seq,par}}$ ideally is the best sequential, parallel *algorithm*
  - these may differ, while here the DAGs are kept the same(!);

Scheduling in realistic models

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

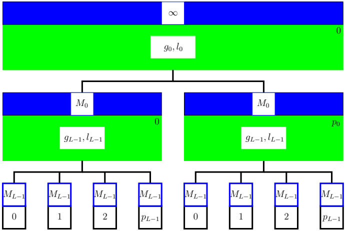
Remarks:

- $T_{\text{seq,par}}$ ideally is the best sequential, parallel *algorithm*
  - these may differ, while here the DAGs are kept the same(!);
- similar for vertical data movement (NP-hard to $n^{1-\epsilon}$, $\forall\epsilon > 0$),
  - with and without **partial computations**.

Ref.: DAG Scheduling in the BSP Model, Papp, Anegg, Y., SOFSEM '25, **best paper award**
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. (submitted)

Scheduling in realistic models

Modern systems: pervasively **NUMA**. Consider Multi-BSP (Valiant '08, '11):



Theorem

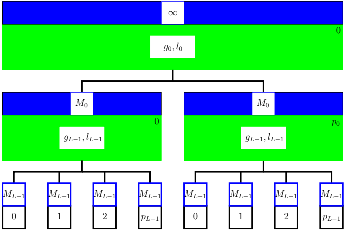
Given an optimal BSP solution to ϵ-balanced hypergraph partitioning and a two-level Multi-BSP ($d = 2$). Optimal hierarchical assignment is NP-hard for $p_2 > 2$, and results in a g_1-approximation (up to $\frac{p_1 - 1}{p_1} g_1$).

(See paper for $d > 2$).

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

Scheduling in realistic models

Modern systems: pervasively **NUMA**. Consider Multi-BSP (Valiant '08, '11):



Theorem

Given an optimal BSP solution to ϵ-balanced hypergraph partitioning and a two-level Multi-BSP ($d = 2$). Optimal hierarchical assignment is NP-hard for $p_2 > 2$, and results in a g_1-approximation (up to $\frac{p_1 - 1}{p_1} g_1$).

(See paper for $d > 2$). "Pure" scheduling or MPP hardness for $d \geq 2$: TBD.

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

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:

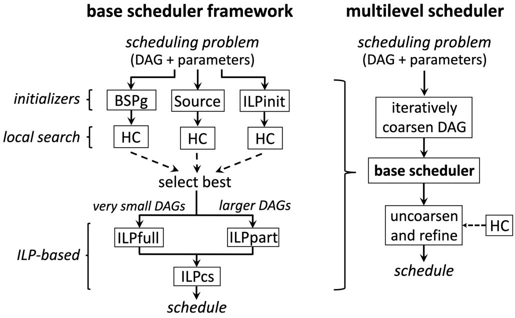
- but only for **small DAGs**: hundreds of nodes, at most.
  - CBC, improves with the use of commercial ILP solvers.
- With **memory constraints** (MPP): 80 nodes, at most (COPT).

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× 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).

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).

Conclusion

We show:

- **hardness** of scheduling under realistic costing (g, l, and NUMA);
- ILP formulations for **optimal scheduling** (horizontal & vertical);
- heuristics (local search, refinement), multi-level, divide & conquer;
- **significant practical gains** over four systems:
  - 2.2–4.2, 1.4–1.7$\times$ faster SpTrsv vs. HDagg, SpMP;
  - 1.2$\times$ faster AI ops, 47% for DeepSeek v3.2 attention.

Conclusion

We show:

- **hardness** of scheduling under realistic costing (g, l, and NUMA);
- ILP formulations for **optimal scheduling** (horizontal & vertical);
- heuristics (local search, refinement), multi-level, divide & conquer;
- **significant practical gains** over four systems:
  - 2.2–4.2, 1.4–1.7$\times$ faster SpTrsv vs. HDagg, SpMP;
  - 1.2$\times$ faster AI ops, 47% for DeepSeek v3.2 attention.

Next:

- maxBSP (Valiant '90) and **SSP** with arbitrary staleness k
  - additional $\approx$ 20% gain for SpTrsv ($k = 1$), paper soon;
- Optimal Multi-BSP **partitioning**, for training & online inference:
  - PP device reduction (0.67$\times$), EP for **throughput**.

Conclusion

We show:

- **hardness** of scheduling under realistic costing (g, l, and NUMA);
- ILP formulations for **optimal scheduling** (horizontal & vertical);
- heuristics (local search, refinement), multi-level, divide & conquer;
- **significant practical gains** over four systems:
  - 2.2–4.2, 1.4–1.7$\times$ faster SpTrsv vs. HDagg, SpMP;
  - 1.2$\times$ faster AI ops, 47% for DeepSeek v3.2 attention.

Next:

- maxBSP (Valiant '90) and **SSP** with arbitrary staleness k
  - additional $\approx$ 20% gain for SpTrsv ($k = 1$), paper soon;
- Optimal Multi-BSP **partitioning**, for training & online inference:
  - PP device reduction (0.67$\times$), EP for **throughput**.

It's open –Apache 2.0– and free to contribute, collaborate:

`https://github.com/Algebraic-Programming/OneStopParallel`

Backup slides

Backup slides

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, Vlačić, 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).

Humble and Hero Programming

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

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

Humble and Hero Programming

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

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

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

Humble and Hero Programming

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

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

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

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

Humble and Hero Programming

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

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

Increasingly many hardware targets,

Humble and Hero Programming

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

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

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

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

Increasingly many hardware targets,

increasingly heterogeneous hardware:

Humble and Hero Programming

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

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

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

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

Increasingly many hardware targets,

increasingly heterogeneous hardware:

a software productivity crisis is looming

Compilers, libraries, and applications

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

Compilers, libraries, and applications

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

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

Compilers, libraries, and applications

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

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

Compilers, libraries, and applications

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

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

Compile humble code to new architectures:

Compilers, libraries, and applications

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

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

Compile humble code to new architectures:

- compiler would need to 'think' about algorithms;

Compilers, libraries, and applications

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

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

Compile humble code to new architectures:

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

Compilers, libraries, and applications

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

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

Compile humble code to new architectures:

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

Compilers, libraries, and applications

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

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

Compile humble code to new architectures:

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

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

- rewrite software or use older/slower hardware?

Compilers, libraries, and applications

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

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

Compile humble code to new architectures:

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

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

- rewrite software or use older/slower hardware?

Many existing software, many workload domains, many architectures:

Compilers, libraries, and applications

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

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

Compile humble code to new architectures:

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

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

- rewrite software or use older/slower hardware?

Many existing software, many workload domains, many architectures:

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

Basics

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

Basics

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

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

1) containers: scalars, vectors, and matrices;

2) structures:

3) primitives:

Basics

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

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

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

Basics

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

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

1) containers: scalars, vectors, and matrices;

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

3) primitives: eWiseApply, reduction into scalar or vector (fold), mxv.

Basics

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

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

1) containers: scalars, vectors, and matrices;

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

3) primitives: eWiseApply, reduction into scalar or vector (fold), mxv.

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

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

Basics

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

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

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

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

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

Elements may be **any POD type**

```
grb::Vector< std::pair< int, double > > pairs( n );
```

Basics

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

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

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

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

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

Elements may be **any POD type**, containers have **capacities**

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

Basics

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

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

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

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

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

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

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

Basics

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

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

Basics

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

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

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

- enables auto-parallelisation and other automatic optimisations.

Basics

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

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

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

- enables auto-parallelisation and other automatic optimisations.

More complex structures may be **composed**:

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

Basics

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

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

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

- enables auto-parallelisation and other automatic optimisations.

More complex structures may be **composed**:

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

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

Basics

Algebraic primitives operate on algebraic containers:

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

Basics

Algebraic primitives operate on algebraic containers:

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

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

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

Basics

Algebraic primitives operate on algebraic containers:

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

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

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

Basics

Algebraic primitives operate on algebraic containers:

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

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

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

Basics

Algebraic primitives operate on algebraic containers:

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

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

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

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

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

Basics

Algebraic primitives operate on algebraic containers:

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

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

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

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

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

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

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

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

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

Algebraic type traits transfer to richer algebraic structures:

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

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

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

Algebraic type traits transfer to richer algebraic structures:

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

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

Algebraic type traits

Algebraic type traits: compile-time introspection of algebraic info

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

Algebraic type traits transfer to richer algebraic structures:

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

  therefore

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

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

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

Algebraic type traits

Algebraic type traits help

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

Algebraic type traits

For example, **algebraic type traits prevent**

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

Algebraic type traits

For example, **algebraic type traits prevent**

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

Algebraic type traits

For example, **algebraic type traits prevent**

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

Algebraic type traits

For example, **algebraic type traits prevent**

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

    parallelisation requires identity *and* associativity!

    foldl (alpha , x, addMon); ✓

Algebraic type traits

For example, **algebraic type traits prevent**

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

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

Algebraic type traits

For example, **algebraic type traits prevent**

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

    foldl (alpha, x, addMon); ✓

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

clear error messages.

Algebraic type traits

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

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

Algebraic type traits

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

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

Algebraic type traits

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

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

Algebraic type traits

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

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

Algebraic type traits

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

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

Algebraic type traits

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

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

Algebraic type traits

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

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

Algebraic type traits

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

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

Algebraic type traits

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

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

These optimisations are applied at compile-time,

Algebraic type traits

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

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

These optimisations are applied at compile-time,

without requiring programmer knowledge or intervention.

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

Performance semantics

Every backend defines **performance semantics**:

- work;

Performance semantics

Every backend defines **performance semantics**:

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

Performance semantics

Every backend defines **performance semantics**:

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

Performance semantics

Every backend defines **performance semantics**:

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

Performance semantics help

- guide programmers to express the best possible algorithm;

Performance semantics

Every backend defines **performance semantics**:

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

Performance semantics help

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

Performance semantics

Every backend defines **performance semantics**:

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

Performance semantics help

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

Performance semantics

Every ALP program can be **systematically costed**:

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

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

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

Performance semantics

Every container has **memory use semantics**:

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

Performance semantics

Every container has **memory use semantics**:

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

Capacities are **optional** during container construction:

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

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

Performance semantics

Every container has **memory use semantics**:

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

Capacities are **optional** during container construction:

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

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

Capacities:

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

ALP Backends

Compile-time selected **backends**:

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

ALP Backends

Compile-time selected **backends**:

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

Backends may be **composed** to

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

ALP Backends

Compile-time selected **backends**:

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

Backends may be **composed** to

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

Use different backends **without ever changing the ALP programs**(!)

ALP Backends

Compile-time selected **backends**:

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

Backends may be **composed** to

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

Use different backends **without ever changing the ALP programs**(!)

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

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

ALP Backends

Compile-time selected **backends**:

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

Backends may be **composed** to

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

Use different backends **without ever changing the ALP programs**(!)

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

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

ALP Backends

Compile-time selected **backends**:

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

Backends may be **composed** to

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

Use different backends **without ever changing the ALP programs**(!)

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

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

ALP Backends

Compile-time selected **backends**:

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

Backends may be **composed** to

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

Use different backends **without ever changing the ALP programs**(!)

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

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

Performance

For the nonblocking backend, expect:

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

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

Performance

For the nonblocking backend, expect:

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

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

Performance

For the nonblocking backend, expect:

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

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

Performance

For the nonblocking backend, expect:

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

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

- 5–19% faster than GSL (sequential);

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

Performance

For the nonblocking backend, expect:

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

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

- 5–19% faster than GSL (sequential);
- 0.25–2.43$\times$ vs. blocking ALP/GraphBLAS;
- 0.49–8.78$\times$ vs. SuiteSparse:GraphBLAS;
- 2.87–7.06$\times$ vs. Eigen– which **also performs loop fusion**.

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

Performance

For the nonblocking backend, expect:

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

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

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

Performance

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

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

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

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

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

Performance

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

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

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

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

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

- parallel I/O: $2.83\times$

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

Performance

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

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

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

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

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

- parallel I/O: $2.83\times$; k-hop BFS: 2.09–$2.53\times$

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

Performance

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

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

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

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

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

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

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

Performance

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

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

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

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

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

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

Distributed performance depends on **data distribution**.

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

Performance

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

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

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

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

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

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

Distributed performance depends on **data distribution**.

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

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

Performance

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

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

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

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

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

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

Distributed performance depends on **data distribution**.

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

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

Performance

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

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

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

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

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

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

Distributed performance depends on **data distribution**.

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

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

Sparse solvers in ALP

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

Sparse solvers in ALP

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

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

Sparse solvers in ALP

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

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

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

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

Sparse solvers in ALP

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

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

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

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

Transition path

Transition paths ensure transparant compatibility:

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

Transition path

Transition paths ensure transparant compatibility:

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

ALP v0.8 comes with the following prototypes:

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

Transition path

Transition paths ensure transparant compatibility:

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

ALP v0.8 comes with the following prototypes:

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

Transition path

Transition paths ensure transparant compatibility:

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

ALP v0.8 comes with the following prototypes:

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

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

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

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

Transition path

Transition paths ensure transparant compatibility:

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

ALP v0.8 comes with the following prototypes:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Performance

ALP interoperability with existing (parallel) frameworks

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

Performance

ALP interoperability with existing (parallel) frameworks

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

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

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

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

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

Performance

ALP interoperability with existing (parallel) frameworks

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

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

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

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

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

Performance

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

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

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

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

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

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

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

Beyond ALP/GraphBLAS

How far can we take this type of programming?

The Alps

The Alps:

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

The ALPs

Algebraic Programming

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

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

The ALPs:

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

Interoperability with existing software:

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

ALP/Dense overview

Classic dense linear algebra:

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

ALP/Dense overview

Classic dense linear algebra:

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

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

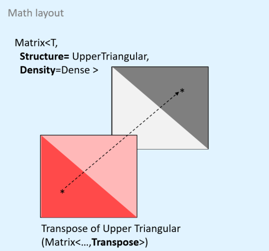
ALP/Dense overview

Classic dense linear algebra:

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

Requires ALP extensions: structures and views

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



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

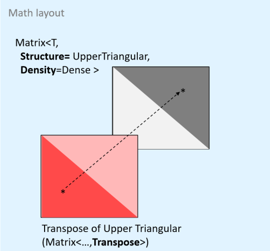
ALP/Dense overview

Classic dense linear algebra:

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

Requires ALP extensions: structures and views

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



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

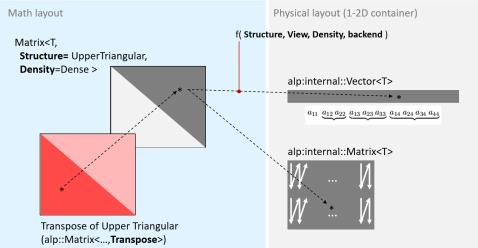
ALP/Dense overview

Classic dense linear algebra:

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

Requires ALP extensions: structures and views

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



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

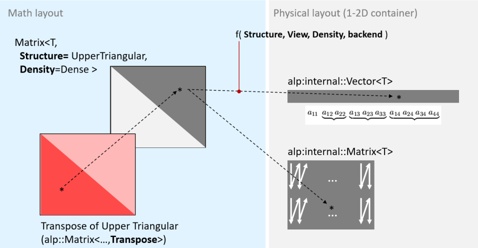
ALP/Dense overview

Classic dense linear algebra:

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

Requires ALP extensions: structures and views

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



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

ALP/Dense overview

Dense computations require **careful tuning**:

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

ALP/Dense overview

Dense computations require **careful tuning**:

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

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

ALP/Dense overview

Dense computations require **careful tuning**:

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

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

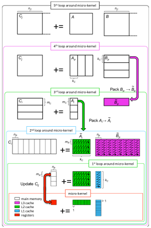
- high-level MLIR dialect introducing, e.g., algebraic structures

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

ALP/Dense overview

Dense computations require **careful tuning**:

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



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

ALP/Dense overview

Dense computations require **careful tuning**:

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

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

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

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

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

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

ALP/Dense overview

Dense computations require **careful tuning**:

1) lazy-evaluate ALP primitives (alike to nonblocking)
2) when pipelines execute, instead first translate to MLIR;
   - high-level MLIR dialect introducing, e.g., algebraic structures
3) BLIS-like approach to optimise MLIR or dispatch to BLAS:
   - use offline (auto-)tuning, **once** per new architecture ✓
4) threads and/or processes **individually** follow steps 2-3;
   - data distribution and parallelisation controlled by ALP.

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

ALP/Dense overview

Dense computations require **careful tuning**:

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

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

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

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

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

4) threads and/or processes **individually** follow steps 2-3;

- data distribution and parallelisation controlled by ALP.

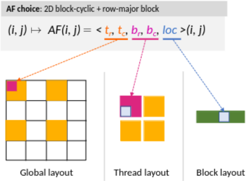
Between JIT and AOT: **delayed compilation or dispatching**

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

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

ALP/Dense implementation

ALP/Dense employs **block-wise storage** via composable xMFs:




```
#pragma omp parallel
{
  thread = config::OMP::current_thread_ID();
  th_coo = da.getThreadCoords( thread );
  blk_grd = da.getLocalBlockGridDims( th_coo );

  for( auto &blk_coo: blk_grd ) {
    // Assuming shmem::seq == reference, Aloc has
   type
    // Matrix< ... , reference >
    auto Aloc = internal::get_view< shmem::seq >(
        A, th_coo, blk_coo
      );
      // Calls func on L.21
      eWiseMul( Aloc, alpha, beta, Sr );
  } }
```

Fig.: storage illustration (left) and code sketch for $A \otimes \alpha \oplus \beta$ (right)

ALP/Dense:

- performs **compile-time simplification** of telescopic composed xMFs
- dispatches block computations to **sequential** hand-tuned BLAS
- performs **NUMA-aware allocation** of thread-owned blocks
- **blocking execution** within threads

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

ALP/Dense evaluation

Expressivity: the API captures the following LA algorithms

- Householder tridiagonalisation, Cholesky, LU, QR, and more

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

ALP/Dense evaluation

Expressivity: the API captures the following LA algorithms

- Householder tridiagonalisation, Cholesky, LU, QR, and more

Performance: 2× HiSilicon Kunpeng 920-**48**26 (4 NUMA domains)

- ALP/Dense (block size 128) compared to Netlib LAPACK;
- LAPACK linked to hand-tuned shared-memory parallel BLAS.

ALP/Dense evaluation

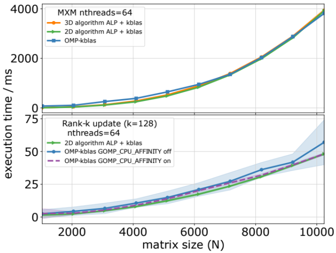
Expressivity: the API captures the following LA algorithms

- Householder tridiagonalisation, Cholesky, LU, QR, and more

Performance: 2× HiSilicon Kunpeng 920-4826 (4 NUMA domains)

- ALP/Dense (block size 128) compared to Netlib LAPACK;
- LAPACK linked to hand-tuned shared-memory parallel BLAS.

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



ALP/Dense evaluation

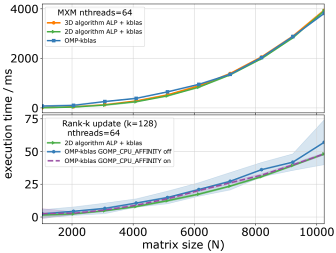
Expressivity: the API captures the following LA algorithms

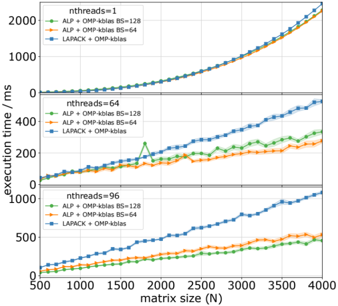
- Householder tridiagonalisation, Cholesky, LU, QR, and more

Performance: 2× HiSilicon Kunpeng 920-4826 (4 NUMA domains)

- ALP/Dense (block size 128) compared to Netlib LAPACK;
- LAPACK linked to hand-tuned shared-memory parallel BLAS.

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





ALP/Dense evaluation

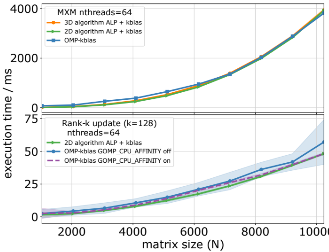
Expressivity: the API captures the following LA algorithms

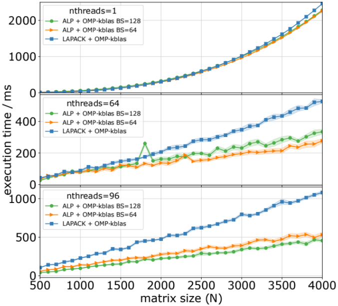
- Householder tridiagonalisation, Cholesky, LU, QR, and more

Performance: 2× HiSilicon Kunpeng 920-**48**26 (4 NUMA domains)

- ALP/Dense (block size 128) compared to Netlib LAPACK;
- LAPACK linked to hand-tuned shared-memory parallel BLAS.

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

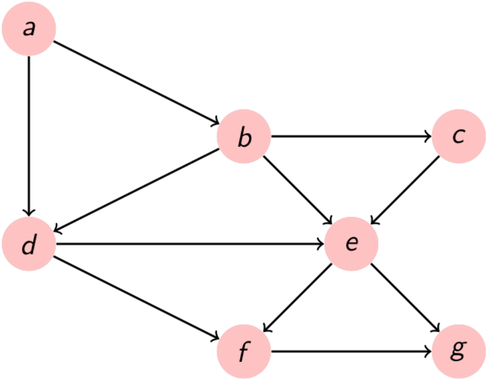




- ALP/Dense NUMA-aware auto-parallelisation: more stable, faster

ALP/Pregel

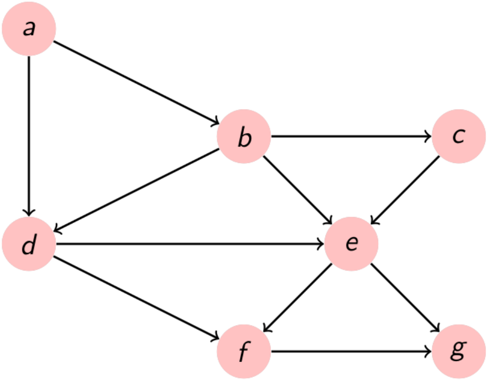
Pregel:



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

ALP/Pregel

Pregel:



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

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

Pregel **page ranking** over directed graphs:

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

Pregel **page ranking** over directed graphs:

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

ALP/Pregel

Pregel **connected components** over undirected graphs:

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

Pregel **page ranking** over directed graphs:

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

ALP/Pregel

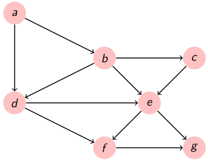
Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



ALP/Pregel

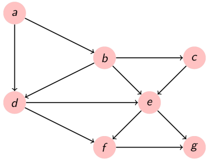
Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



- grb::eWiseLambda executes a vertex program;

ALP/Pregel

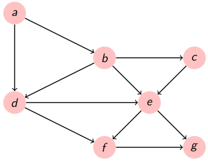
Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



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

ALP/Pregel

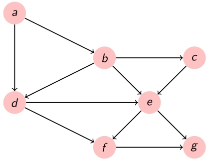
Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



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

ALP/Pregel

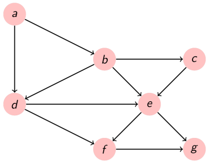
Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



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

ALP/Pregel

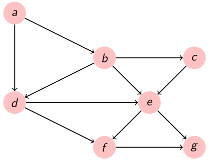
Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



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

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

ALP/Pregel

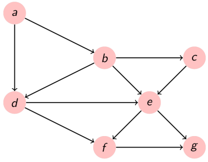
Expanding Pregel programs into ALP/GraphBLAS:

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

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

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

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



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

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

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

ALP/Pregel

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

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

ALP/Pregel

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

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

Using masking to not incur overhead from inactive vertices;

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

ALP/Pregel

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

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

Using masking to not incur overhead from inactive vertices;

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

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

Sequential performance in ms. Compares different page ranking algorithms.

ALP/Pregel

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

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

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

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

ALP/Pregel

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

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

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

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

ALP as a foundational programming model

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

- performance semantics, algebraic type checking, interoperability;

ALP as a foundational programming model

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

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

ALP as a foundational programming model

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

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

ALP as a foundational programming model

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

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

ALP is a *more foundational* programming model than Pregel

ALP as a foundational programming model

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

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

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

- Pregel may be viewed as more humble than ALP.

ALP as a foundational programming model

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

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

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

- Pregel may be viewed as more humble than ALP.

This observations inspires questions

ALP as a foundational programming model

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

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

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

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

- what other humble APIs can ALP efficiently simulate?

ALP as a foundational programming model

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

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

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

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

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

ALP as a foundational programming model

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

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

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

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

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

ALP as a foundational programming model

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

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

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

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

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

One software stack with

ALP as a foundational programming model

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

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

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

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

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

One software stack with

multiple **humble interfaces**

ALP as a foundational programming model

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

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

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

- Pregel may be viewed as more humble than ALP.

This observations inspires questions:

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

One software stack with

multiple **humble interfaces**,

achieving **hero performance**.