Elasticity in Parallel Sparse Triangular Solve

Raphael Sebastian Steiner
Huawei Technologies Co Ltd
Zurich, Switzerland
raphael.steiner@huawei.com

Christos Konstantinos Matzoros
Huawei Technologies Co Ltd
Zurich, Switzerland
christos.konstantinos.matzoros@h-partners.com

Pál András Papp
Huawei Technologies Co Ltd
Zurich, Switzerland
pal.andras.papp@huawei.com

Toni Boehnlein
Huawei Technologies Co Ltd
Zurich, Switzerland
toni.boehnlein@huawei.com

Albert-Jan N. Yzelman
Huawei Technologies Co Ltd
Zurich, Switzerland
albert-jan@yzelman.net

Abstract

We introduce stale synchronous parallel as a mode of execution in parallel sparse triangular linear system solve and present a general directed-acyclic-graph scheduler capable of producing such schedules. Stale-synchronous-parallel schedules allow the overlap of synchronisation and compute which results in a geometric-mean speed-up of 7-30% of our scheduler, ElasticDivide, over state-of-the-art synchronous scheduler GrowLocal on an ARM machine with 48 cores. On an x86 machine with 48 cores, we report geometric-mean speed-ups of 19-60% over SpMP.

CCS Concepts

• **Theory of computation** → *Scheduling algorithms*; *Design and analysis of algorithms*; • **Computing methodologies** → *Linear algebra algorithms*; *Parallel algorithms.*

Keywords

Sparse triangular linear system solve, SpTrSV, SpTrSM, forward- and backward-substitution algorithm, stale-synchronous-parallel algorithm.

ACM Reference Format:
Raphael Sebastian Steiner, Christos Konstantinos Matzoros, Pál András Papp, Toni Boehnlein, and Albert-Jan N. Yzelman. 2026. Elasticity in Parallel Sparse Triangular Solve. In *Proceedings of the 55th International Conference on Parallel Processing (ICPP '26), September 28–October 01, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3832810.3832901

1 Introduction

Sparse triangular solve is an omnipresent operation in computing. Whether it be in engineering, data analytics, artificial intelligence, or scientific computing, systems of equations need to be solved. This typically involves (sparse) triangular solve as part of the solving process directly, through methods like LU, QR, and Cholesky factorisations and Gauß–Seidel, or indirectly through (pre-)conditioning of the system for faster iterative solving methods such as conjugate gradient.

When solving ever bigger problem instances, sparsity plays an evermore important role. It allows one to cut redundant compute, but this comes at the cost of losing structure. This results in fine-grained dependencies and operations which makes sparse triangular solve (SpTrSV) a hard problem to parallelise. Therefore, getting the most performance out of modern multi-core architectures is a challenging endeavour.

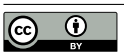
To combat this, SpTrSV performance is often optimised in larger contexts such as linear or symmetric solves where preprocessing allows for the reintroduction of structure. A prominent example of this is the nested dissection technique [5, 6, 21, 23, 29, 32] which concentrates non-zeroes for locality and introduces coarse-grained parallelism [19, 35]. Another important technique is the introduction of so-called supernodes [13, 20, 31, 33, 46, 47, 50]. Supernodes give rise to several optimisations. First, the grouping of operations into supernodes allows for the use of highly optimised dense kernels, and second, the dependency graph becomes coarser which enables dynamically computing a parallel schedule or dynamic dispatching methods as their benefits now outweigh their overhead. Another notable and generally applicable technique is the (recursive) splitting of SpTrSV into two SpTrSV problems of half the size and an easier-to-parallelise SpMV in-between [2, 4, 34, 36, 37].

After applying these structural techniques (if they are feasible), one is left with a parallel scheduling problem. The tasks to be scheduled typically have irregular dependencies and their size can range from whole blocks or supernodes to a couple of floating-point operations. It is this scheduling problem that we address in this paper. More precisely, we consider the fine-grained scheduling problem of parallelising the forward-substitution algorithm, cf. Algorithm 2.1. Nevertheless, our algorithm and the involved ideas may, of course, be applied to any parallel scheduling problem.

To get the most performance out of SpTrSV, a parallel scheduling algorithm must

- balance workloads among cores,
- limit coordination overhead, and
- consider spatial and temporal locality.

Several algorithms to this end have been proposed in the literature. Broadly speaking, they come in two flavours, synchronous and asynchronous, depending on the type of schedule they produce.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ICPP '26, Singapore, Singapore
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2657-6/26/09
https://doi.org/10.1145/3832810.3832901

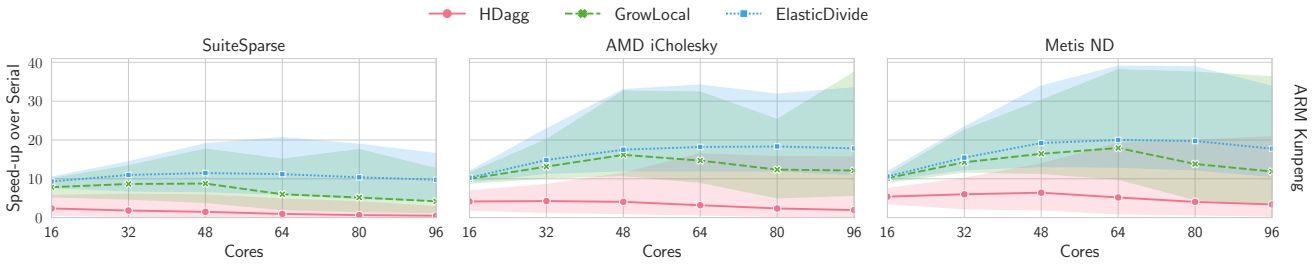


Figure 1.1: Geometric-mean speed-ups over Serial for various data sets, cf. §5.2, on ARM Kunpeng with 10-th to 90-th percentile range shown.

Early algorithms include asynchronous self-scheduling [45] and the synchronous so-called wavefront schedulers [4, 44]. Both approaches suffered from a lot of synchronisation overhead due to frequent synchronisation [40, 43]. In a breakthrough paper [40], Park *et al.* addressed this issue. Their scheduler SpMP combines the coarseness from the grouping of wavefront schedulers with asynchronous compute and point-to-point synchronisation. To further reduce the number of synchronisations, they employed an approximate transitive reduction to get rid of the majority of implied (and thus unnecessary) dependencies. Yılmaz *et al.* [51] were further able to reduce the number of dependencies by forcing the cores to be out-of-sync by only a bounded amount. Additionally, their scheduler ALB takes into account the amount of parallelism available locally in the matrix as the computation moves forward to decide how many of the available cores to use.

On the synchronous front, there have also been improvements. The HDagg scheduler by Zarebavani *et al.* [52] combines successive wavefronts whenever beneficial in order to reduce the number of synchronisations. In contrast, Böhnlein *et al.* [8, 9] moved away from wavefronts and took inspiration from list scheduling [1, 22, 28, 38, 41]. Their scheduler, GrowLocal, interleaves local list scheduling with synchronisation barriers which communicate the completion of rows. This allowed for much longer independent compute, reducing the number of synchronisations by an order of magnitude.

1.1 Our contribution

In this paper, we strike a middle ground between synchronous and asynchronous compute. Namely, we employ for the first time stale synchronous parallel [12, 14] as the mode of execution in sparse triangular solve. This execution model explicitly overlaps synchronisation and compute by forcing inter-core dependencies to be elongated. In other words, the information computed by one core can only be used on another core after several synchronisation events. Since the information is not immediately required after a synchronisation event, the synchronisation can be overlapped with other compute based on already locally available information. In the case of synchronisation-heavy workloads with many threads, we show that this can result in a 2× geometric-mean speed-up over synchronous execution. On architectures with higher number of NUMA-domains, this can go up to 4.5×. Even when the architecture is more uniform and the synchronisation events are few, it leads to improvement in the double digit percentages.

Naturally, stale synchronous parallel puts additional strain on the schedule to be generated and therefore also on the scheduler. Nevertheless, we demonstrate empirically that the computational graphs associated to an SpTrSV of a matrix are elastic enough to allow for such schedules. Indeed, our scheduler *ElasticDivide* produces about the same number of synchronisation barriers (0.85× to 1.28×) as the state-of-the-art synchronous scheduler GrowLocal [8] when appropriately normalised, yet overlaps compute with synchronisation. In Figure 1.1, we depict geometric-mean speed-ups over Serial on an ARM Kunpeng machine with 48 cores and two NUMA-domains per socket. The three plots correspond to three data sets which are derived from the SuiteSparse Matrix Collection [16]. SuiteSparse consists of matrices directly taken from the data set, and the other two data sets are application oriented with the matrices having been preprocessed with an approximate-minimal-degree ordering and incomplete Cholesky using Eigen [24], respectively with nested dissection using Metis [29]. The reported geometric-mean speed-ups range from 3% on small core counts to over 100% on large core counts over the second best measured algorithm GrowLocal. On 48 cores (single socket), the geometric-mean speed-ups are 30%, 7%, and 17%, respectively. SpMP, being x86-specific, is missing from Figure 1.1. On 48 cores on an AMD EPYC machine, we achieve geometric-mean speed-ups of 19%, 60%, and 26%, respectively, over SpMP with geometric-mean speed-ups reaching over 100% on smaller core counts.

In summary, in this paper, we

- propose and analyse solving SpTrSV problems via stale synchronous parallelisation,
- introduce ElasticDivide, a stale-synchronous-parallel scheduler for general directed acyclic graphs, and
- show their effectiveness on non-uniform memory architectures.

1.2 Overview

We describe the stale-synchronous-parallel scheduling problem in §2. The ElasticDivide scheduler is presented in detail in §3. In §4, we describe the implementation of the kernel and the barrier. In §5 and §6, we describe our evaluation procedure and evaluate the algorithms, followed by a discussion in §7.

Acknowledgements

We would like to thank Weifeng Liu, Olaf Schenk, Xiaoye Li, Dimosthenis Pasadakis, Lorenzo Migliari, and Kiril Dichev for stimulating conversations on this and surrounding topics. We also thank the reviewers for valuable comments and suggestions.

2 Preliminaries

2.1 Problem description

In sparse triangular solve (vector), one is given a sparse invertible matrix $\boldsymbol{A} = (A_{i,j})_{i,j=1,\dots,n} \in \mathbb{R}^{n\times n}$ and a vector $\boldsymbol{b} = (b_i)_{i=1,\dots,n} \in \mathbb{R}^n$ and seeks to compute the vector $\boldsymbol{x} = (x_i)_{i=1,\dots,n} \in \mathbb{R}^n$ such that

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.1}$$

holds. The standard algorithm to solve for $\boldsymbol{x}$ is the *forward-substitution algorithm* if the matrix $\boldsymbol{A}$ is lower triangular, respectively the *backward-substitution algorithm* if $\boldsymbol{A}$ is upper triangular. Assume from now on that $\boldsymbol{A}$ is lower triangular. The forward-substitution algorithm is described in Algorithm 2.1.

Algorithm 2.1: Forward-substitution algorithm

Data: An invertible lower-triangular matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and a vector $\boldsymbol{b} \in \mathbb{R}^n$.
Result: A vector $\boldsymbol{x} \in \mathbb{R}^n$ such that $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$.

```
1 for i = 1, ..., n do
2     t ← b_i
3     for j = 1, ..., i − 1 with A_{i,j} ≠ 0 do
4         t ← t − A_{i,j} x_j
5     x_i ← t / A_{i,i}
```

In contrast to the dense case, when the matrix $\boldsymbol{A}$ is sparse, there is opportunity to parallelise the outer for-loop, cf. Line 1. Though, one has to be careful as there are dependencies. The i-th iteration depends directly on the j-th iteration if and only if $A_{i,j} \neq 0$ and $i \neq j$. More generally, the i-th iteration depends on the j-th iteration if and only if there is a sequence $i = \ell_0 > \ell_1 > \cdots > \ell_m = j$ with $A_{\ell_k,\ell_{k+1}} \neq 0$ for $k = 0, \dots, m-1$ for some $m \geq 1$. We can and shall capture this in a directed acyclic graph $G_{\boldsymbol{A}} = (V_{\boldsymbol{A}}, E_{\boldsymbol{A}})$, where

$$V_{\boldsymbol{A}} = \{1, \dots, n\}, \text{ and} \tag{2.2}$$

$$E_{\boldsymbol{A}} = \{(j,i) \in V_{\boldsymbol{A}} \times V_{\boldsymbol{A}} \mid A_{i,j} \neq 0\}\,. \tag{2.3}$$

We may also attach a (vertex-)weight to $G_{\boldsymbol{A}}$ modelling the execution time of the i-th iteration corresponding to a vertex $i \in V_{\boldsymbol{A}}$. A natural choice as this weight is the number of non-zero elements the corresponding row, that is

$$\begin{aligned} \omega_{\boldsymbol{A}} : V_{\boldsymbol{A}} &\to \mathbb{Z}_{>0} \\ i &\mapsto \sharp\{j \in V_{\boldsymbol{A}} \mid A_{i,j} \neq 0\}\,. \end{aligned} \tag{2.4}$$

This is also equal to the in-degree plus one in the graph representation.

From here on out, we work with the graph representation $G = G_{\boldsymbol{A}}$ of the sparse lower triangular matrix $\boldsymbol{A}$ and formulate the problem as a scheduling problem in an execution model described in the next section.

2.2 Stale synchronous parallel

Stale synchronous parallel is a computational model extending the bulk-synchronous-parallel model [48, 49]. It was formally introduced by Cui *et al.* [14] and originally stems from a paper by Cipar *et al.* [12] in machine learning. Similar to the bulk synchronous parallel, stale synchronous parallel operates in 'rounds' which are called *supersteps*, but, unlike bulk synchronous parallel, cores may move the computation ahead into subsequent supersteps as long as the maximal difference in the superstep index remains bounded by a fixed quantity called the *staleness*. Formally, a compute schedule in the stale-synchronous-parallel model may be defined as follows.

Definition 2.1. *A* stale-synchronous-parallel schedule *of* staleness $s \in \mathbb{Z}_{>0}$ *of a directed acyclic graph* $G = (V, E)$ *to a set of cores* P *consists of a pair* (π, σ) *of maps* $\pi : V \to P$ *and* $\sigma : V \to \mathbb{Z}_{\geq 0}$ *satisfying*

$$\forall (v,w) \in E :\ \sigma(v) + s \cdot \delta_{\pi(v),\pi(w)} \leq \sigma(w)\,, \tag{2.5}$$

where $\delta_{i,j}$ *is the Kronecker delta. The map* π *maps vertices to cores and the map* σ *maps vertices to* supersteps.

Remark 2.2. *The definition of a stale-synchronous-parallel schedule of staleness* $s = 1$ *is the same as a bulk-synchronous-parallel schedule.*

Remark 2.3. *A stale-synchronous-parallel schedule of staleness* s *is naturally a stale-synchronous-parallel schedule of staleness* t *such that* $1 \leq t \leq s$*, in particular also a bulk-synchronous-parallel schedule.*

The use of the word 'stale' in the name 'stale synchronous parallel' is perhaps unfortunate and stems from the first paper [12], where it was used in machine-learning model training. Workers were allowed to compute model updates based on a mildly outdated (stale) model, which allowed for some asynchronicity whilst (provably) maintaining convergence and quality guarantees. Only later [14], it was realised that the 'data' need not be outdated (or stale), but instead having 'long' dependencies suffices.

The stale-synchronous-parallel model has since found many applications, such as recommendation systems [17], matrix factorisation [26], topic modeling [26], and training for deep learning [53]. We also note that overlapping computation and communication, a key motivation for the stale synchronous parallel model, had already been identified as a core feature of the bulk synchronous parallel model [48].

3 The ElasticDivide scheduler

Our scheduling algorithm ElasticDivide is an extension of the algorithm *GrowLocal* presented in [8]. A sketch of ElasticDivide is given by Algorithm 3.1. The main novelty of our algorithm is the adaptation to a stale-synchronous-parallel schedule of staleness 2, cf. Definition 2.1. To achieve this, we had to innovate and add two key features:

(i) a delay in resolving dependencies across cores, and
(ii) strike a balance between assigning vertices in the current superstep and the next.

The algorithm proceeds by assigning vertices in the order of supersteps. During each superstep it assigns *assignable* vertices to cores, cf. *Rule I.* Rather than going through time steps to keep a

Algorithm 3.1: Sketch of ElasticDivide scheduler

Data: A vertex-weighted directed acyclic graph $G = (V, E, \omega)$ and cores $P = \{1, 2, \ldots, k\}$.
Result: Assignments of vertices to cores $\pi : V \to P$ and superstep $\sigma : V \to \mathbb{Z}_{\geq 0}$.
Rule I: A vertex v is *assignable* to core p and superstep $s \in \mathbb{Z}_{\geq 0}$ if and only if

$$\forall (w, v) \in E : \sigma(w) + 2 \cdot \delta_{\pi(w),p} \leq s \,.$$

Rule II: Vertices are prioritised according to
(i) core exclusivity, and then
(ii) smallest ID.

```
1  s ← 0
2  while not all vertices are assigned do
3      α ← 20
4      while true do
           // I. Assign vertices to cores
5          assign up to α assignable vertices to core 1 and
            superstep s with Rules I & II
6          Ω_1 ← total newly assigned weight to core 1
7          for core p = 2, ..., k in order do
8              Ω_p ← 0
9              while Ω_p ≉ Ω_1 do
10                 if ∃ assignable vertex v to core p and
                    superstep s then
11                     assign vertex v to core p and superstep s
                        with Rules I & II
12                     Ω_p ← Ω_p + ω(v)
13                 else break
           // II. Score assignments
14         β ← (Σ_p Ω_p) / (max_p Ω_p + 2000)
15         exit ← true
16         if the score β is high enough then
17             deem current assignments as worthy
18             exit ← false
19         if not enough assignable vertices for next superstep
            then
20             exit ← true
21         undo vertex assignments with superstep s
22         α ← (3/2) α
23         if exit then
24             redo last worthy assignments
25             s ← s + 1
26             break inner loop
```

work balance between cores, as you would in (barrier) list schedulers [1, 22, 28, 38, 39, 41], the algorithm has a guess α as to how long it may compute during a superstep and fills up each core up to that limit in order. This allows the algorithm to prioritise locality of the execution according to *Rule II*. The second nature of *Rule I* is that it extends compute during a single superstep by assigning more restricted vertices first. Once no further assignments are being made, the assignments in their totality are measured by a scoring function, cf. Line 14. If the score is high enough, the assignments are considered successful and another attempt with increased α is initiated. The increase of α is exponential as to amortise the costs of retrying assignments of the current superstep. Starting with a small α and gradually increasing it allows the algorithm to keep a work balance, but also maximise independent compute. The balance between work balance and independent compute is determined by the score β. The scoring function is taken from [8] and is inversely proportional to the execution time of the current superstep in a bulk-synchronous-parallel model without communication [48] scaled up to the whole graph. The constant 2000 thus plays the role of the synchronisation cost. Albeit our algorithm produces a stale-synchronous-parallel schedule with staleness 2 rather than a bulk-synchronous-parallel schedule and can thus overlap synchronisation and compute, we find that the scoring function nevertheless does a good job in balancing parallelism and the length of independent compute.

Returning to the flow of the algorithm, the superstep attempts are interrupted when either the score β starts to drop or it is deemed that there are no longer sufficient *assignable* vertices left for the following superstep. At this point, the best assignment measured by the score β is chosen as the final assignment of the current superstep and the algorithm moves onto the next superstep.

The algorithm is efficiently implemented using sorted double-ended queues for the globally ready queues of vertices which are *assignable* to all cores for a given superstep, and heap for locally ready queues of vertices which are *assignable* to a *single* given core and given superstep. As a result, the presented algorithm runs in almost linear time under reasonable assumptions. We refer to [8, Appendix B] for a proof and §6.5 for empirical data.

At last, we also remark on the choice of staleness 2. It is the smallest number that allows the overlap of compute and synchronisation. Larger numbers would only be beneficial if the compute is too small compared to the synchronisation. We find this to be unlikely the case. Moreover, in our initial testing, we have found that larger staleness negatively affects the schedule quality, thus degrading the overall performance. Hence, we chose staleness equal to 2.

4 SpTrSV implementation

4.1 Synchronisation mechanism

A stale-synchronous-parallel schedule of staleness $s \geq 2$ allows one to overlap synchronisation with computation. To this end, we implemented a weak (or fuzzy) barrier [25] which separates the `arrive` and `wait` operations.

In our barrier implementation, each core has its own flag to signal that it has arrived at the barrier. The flag-owning core has exclusive write access to its flag. All other cores may read this flag to check whether the flag-owning core has arrived at the barrier. This allows us to avoid read-modify-write operations. In addition, our barrier makes use of caching [30, 42], that is, every core has their own local cache of the flags of all other cores. This reduces contention and communication.

To make the most out of the caching mechanism, to save memory, and to reduce code complexity, we combine all barriers into one by having a counter as the flag. This counter records the superstep which the core is currently processing or is waiting to start processing, meaning the core has completed the computation of all previous supersteps. This greatly benefits the caching mechanism as the flag of a core which is (far) ahead in terms of supersteps only needs to be checked once until the probing core has caught up. In fact, the probing core not only can catch up but also move ahead of the cached value due to the staleness before having to read again the flag of the same core.

The `wait` operation was implemented using busy waiting.

For other kinds of barriers, we refer to the survey paper [27] and the references therein.

4.2 Stale-synchronous-parallel SpTrSV kernel

Our stale-synchronous-parallel SpTrSV kernel solves a lower triangular linear system in parallel according to a stale-synchronous-parallel schedule (π, σ) of staleness 2, cf. §2.2. We parallelise the kernel using OpenMP [15] threads. Each thread runs on its own physical core and represents a core p in the schedule. In each superstep s, every core p executes the computations associated to the rows whose corresponding vertex v has been assigned to said processor and superstep, i.e., $v \in \pi^{-1}(\{p\}) \cap \sigma^{-1}(\{s\})$, in ascending order. The supersteps are surrounded with the weak barrier mechanics from §4.1, that is, the `wait` operation at the beginning of the superstep and `arrive` at the end of the superstep. We note that the `wait` operation only needs to be invoked if there is something to compute for the core during the superstep. If there is not, the `wait` can be safely skipped. As supersteps are sometimes empty due matrix-parallelism limitations and algorithm choices, this leads to improvements due to the implemented caching mechanism of the barrier, cf. §4.1.

Algorithm 4.1: Stale-synchronous-parallel SpTrSV kernel

Data: An invertible lower-triangular matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, a vector $\boldsymbol{b} \in \mathbb{R}^n$, a set of cores $P = \{1, 2, \ldots, k\}$, and a stale-synchronous-parallel schedule (π, σ) for $G_{\boldsymbol{A}}$ of staleness 2, cf. §2.2.

Result: A vector $\boldsymbol{x} \in \mathbb{R}^n$ such that $\boldsymbol{Ax} = \boldsymbol{b}$.

```
for core p ∈ P do in parallel
    for s = 0, 1, ..., max σ(V_A) do
        if π⁻¹({p}) ∩ σ⁻¹({s}) ≠ ∅ then wait(p, s − 2)
        for i ∈ π⁻¹({p}) ∩ σ⁻¹({s}) do
            t ← b_i
            for j = 1, ..., i − 1 with A_{i,j} ≠ 0 do
                t ← t − A_{i,j} x_j
            x_i ← t / A_{i,i}
        arrive(p, s)
```

5 Experimental setup

This section presents the benchmarks and setup used for our experiments. In large part, we follow the experimental setup of [8, 9] and [52].

The implementation of our algorithm and the test suite are available open source within the *OneStopParallel* scheduling framework on Github [7].

5.1 Methodology

Our own scheduling algorithm is evaluated using the SpTrSV kernel described in §4.2. The kernel is parallelised through the OpenMP library [15], with `OMP_PROC_BIND=close` and `OMP_PLACES=cores` as the settings. The scheduler GrowLocal is implemented in a similar fashion with the appropriate bulk-synchronous-parallel kernel implementation.

The baseline schedulers HDagg and SpMP are evaluated using the Sympiler framework of Chesmi *et al.* [10, 11]. This contains the original implementation of HDagg [52] and has SpMP [40] already integrated into it.

For every scheduler and problem instance, we repeat the SpTrSV execution 100 times in order to obtain a reliable estimation of the running time. We also add two untimed executions beforehand in order to ensure that the measurements all happen in a 'hot' state of the system. Between two consecutive SpTrSV executions, the right-hand-side vector $\boldsymbol{b}$ is always reset to the all-ones vector.

The test suite of our kernel inherently implements the process above. With minimal modifications, we also adjusted the runtime measurement module of Sympiler to follow the same methodology.

All scheduling algorithms (ours and the baselines) are implemented in C++ and were compiled with GCC (11.4.0 or 11.5.0) using the optimisation flag `-O3`. The running times of the algorithms are measured using the high-resolution clock in C++ (available in `std::chrono`).

5.2 Data sets

We evaluate the SpTrSV scheduling algorithms on several different data sets. The data sets all originate from the SuiteSparse Matrix Collection [16]. This collection contains a diverse set of matrices from a variety of applications, and it is the prominent evaluation benchmark in previous works on SpTrSV scheduling [8, 9, 40, 52]. Following in large part [8, 9] and [52], one of our data sets consists of matrices directly from SuiteSparse. The other two data sets consist of modified versions of the same matrices in order to align more closely with applications and previous benchmarks.

5.2.1 SuiteSparse. This data set is taken from [8, 9] and consists of the lower triangular part of real symmetric positive definite matrices from the SuiteSparse Matrix Collection [16]. It encompasses 33 matrices. The SpTrSV problem associated with each matrix involves at least 2 million floating point operations and has an average wavefront size of at least 44.

5.2.2 SuiteSparse Eigen incomplete Cholesky (iChol). Our second data set is also taken from [8, 9] and consists of lower triangular matrices obtained from an incomplete Cholesky decomposition on the real symmetric matrices from the SuiteSparse data set, see

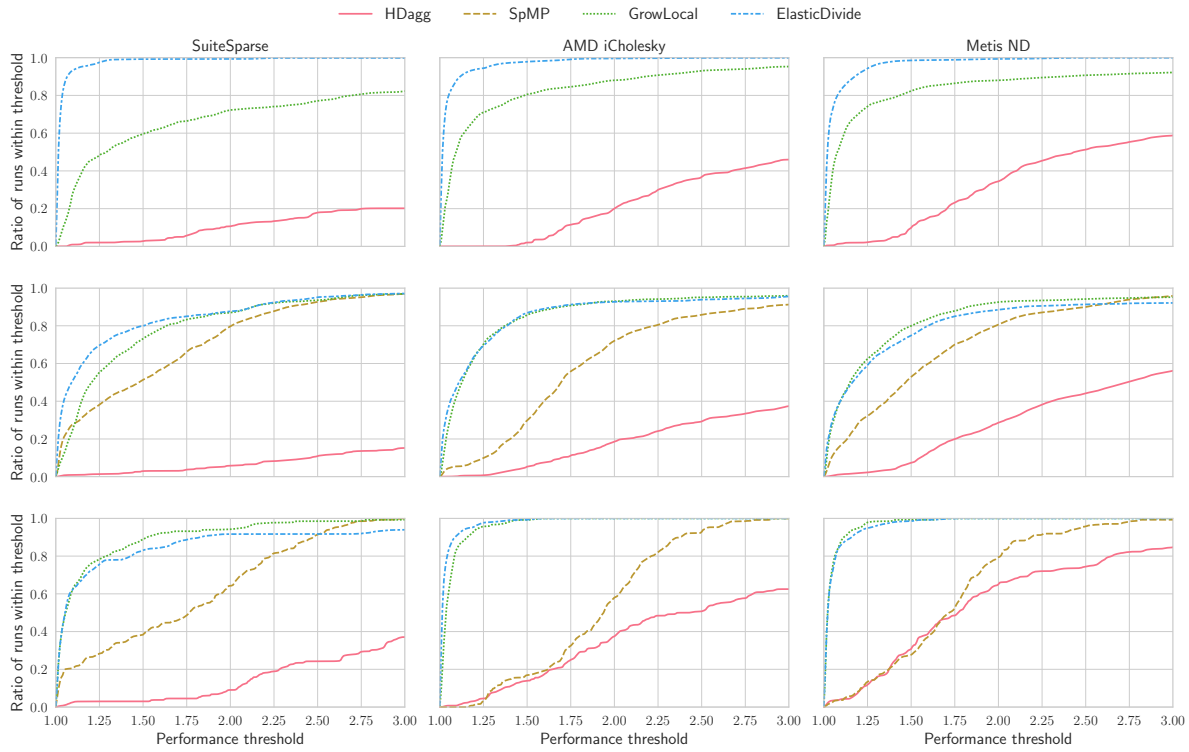


Figure 5.1: Performance profiles of the various SpTrSV scheduling algorithms on a given data set and CPU architecture. In each plot, the x-axis represents a threshold and the y-axis the ratio of runs of a given algorithm which are within this threshold times the best run of the same matrix and core count.

§5.2.1. The incomplete Cholesky decomposition was performed using the 'IncompleteCholesky' method in Eigen [24], which entailed applying its internal reordering method 'AMDOrdering' [3]. We note that the matrix 'bundle_adj' segmentation-faults during this process.

5.2.3 SuiteSparse METIS (METIS). For the final data set, we use the data set considered in [52]. It is once more derived from real symmetric positive definite matrices from the SuiteSparse Matrix Collection. The matrices are preprocessed using the fill-reducing method (nested dissection) of the METIS partitioner [29], which symmetrically permutes the original matrices. After this preprocessing step, the lower triangular part is taken.

The matrices in this data set are representative of SpTrSV workloads in a Gauß–Seidel or a zero-fill-in incomplete Cholesky preconditioned conjugate gradient method for sparse symmetric solve.

5.3 CPU architectures

We evaluated our SpTrSV scheduler on both x86 and ARM architectures. The concrete specifications for the three machines used in the experiments are as follows:

- Huawei Kunpeng 920-4826 (Hi1620) processor (ARM, introduced 2019), with 512 GB memory, theoretical peak memory throughput of 187.7 GB/s, and two sockets with 48 cores each for a total of 96 cores; 2 NUMA domains per socket; kernel version 5.15.0; GCC version 11.4.0.
- AMD EPYC 7763 processor (x86, introduced 2021), with 1024 GB memory, theoretical peak memory throughput of 204.8 GB/s, and two sockets with 64 cores each for a total of 128 cores; 1 NUMA domain per socket; kernel version 5.15.0; GCC version 11.4.0;
- Intel Xeon Gold 6238T processor (x86, introduced 2019), with 192 GB memory, theoretical peak memory throughput of 140.8 GB/s, and two sockets with 22 cores each for a total of 44 cores; 1 NUMA domain per socket; kernel version 5.14.0; GCC version 11.5.0;

6 Evaluation

6.1 Runtime

To summarise our findings, we have condensed in Figure 5.1 the SpTrSV runtimes into performance profiles [18]. We have generated a performance profile for each data set and processor architecture, cf. §5. In each of these performance profiles, we took the best SpTrSV runtime for each core count and matrix combination. For a given algorithm, we then computed the ratio of all SpTrSV from that algorithm that are with a given threshold of the respective best SpTrSV runtime. The resulting curve is then plotted with the x-axis representing the threshold and the y-axis representing the ratio.

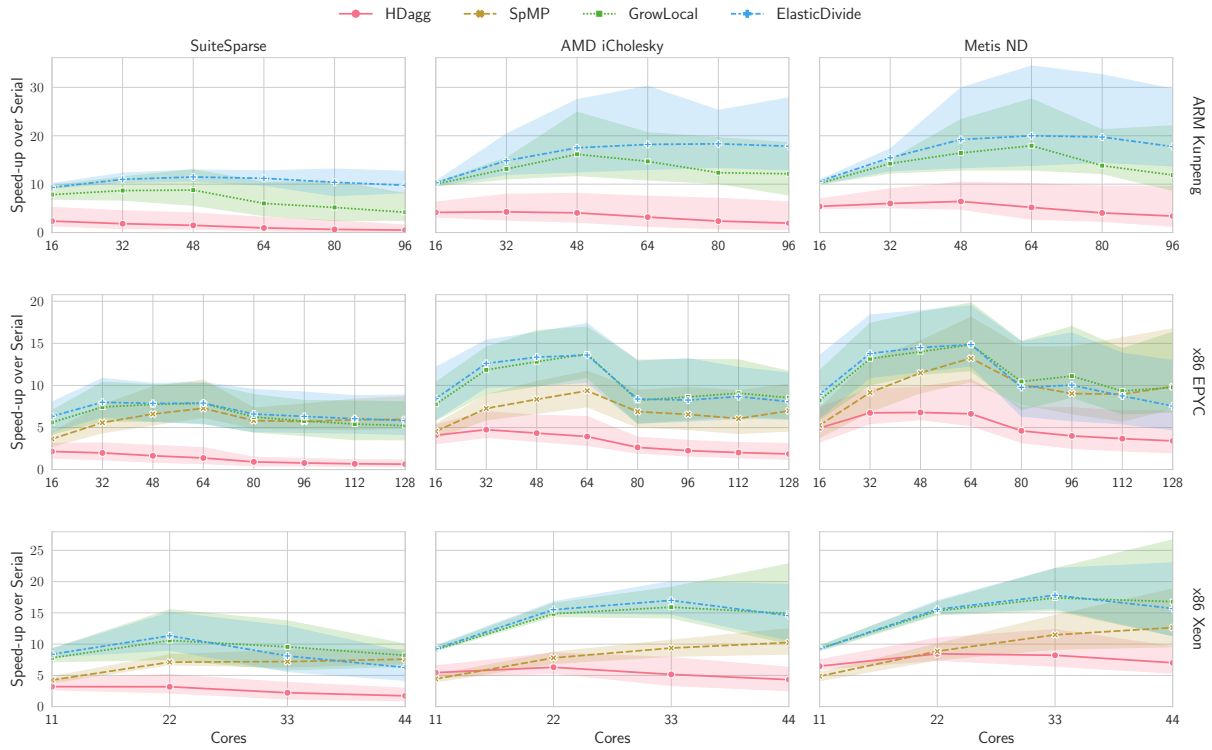


Figure 6.1: Scaling plots depicting the geometric mean speed-up over Serial and interquartile range of each algorithm on a given data set, architecture, and core count.

The closer a curve of a given algorithm is to the top left corner, the better the algorithm is at performing parallel SpTrSV. We can clearly see that the new algorithm ElasticDivide performs exceptionally well on the ARM architecture. Similarly, on x86, ElasticDivide is the most performant, though it shares the top spot with GrowLocal. We explain this with the increased NUMA domains the ARM Kunpeng architecture has, favouring asynchronous compute.

SpMP is a strong contender on the SuiteSparse data set and performs well overall, but ultimately falls a bit short of ElasticDivide and GrowLocal. HDagg struggles with the hard to parallelise SuiteSparse data set and only seems to do well on x86 Xeon architecture with smaller core count.

6.2 Scaling

We investigate how the algorithms scale with the core count by graphing the geometric mean and interquartile range of the speed-up over serial execution. We do this in Figure 6.1 for every pair of architecture and data set, cf. §5.

We once more see that ElasticDivide outperforms all other algorithms on all data sets on the ARM architecture. It reaches a geometric mean speed-up of 20.00 on the Metis data set on 64 cores. This is 12% higher than the speed-up of 17.93 reached by the second best algorithm GrowLocal on the same data set (and architecture).

On the x86 architecture, GrowLocal and ElasticDivide compete closely for the best algorithm. However, as the core count increases, in particular on the SuiteSparse data set, SpMP continues to improve relative to GrowLocal and ElasticDivide and in some cases matches or improves upon their performance.

HDagg does not scale well with the number of cores except for the Metis data set as the nested dissecting reordering works well with the wavefront-gluing method that HDagg employs.

On all architectures, we see a drop in speed-ups at the 80 core count. On the EPYC processor this coincides with a second socket, which is a likely contributor, but on the Kunpeng processor we already require the second socket at 64 cores. Thus, we find the more likely explanation is that the parallelism which is exposed by the matrices and then picked up by the algorithms peters out around that core count, at least for a majority of matrices.

6.3 Synchronisation

In Table 6.1, we present the geometric-mean reduction of the number of synchronisation barriers (weak or strong) relative to the number of wavefronts of the matrix. In other words, the reduction of synchronisation barriers compared to a wavefront scheduler. The numbers we present are from all data sets over a sample of core counts. We note that SpMP is missing in the table as it employs an asynchronous (point-to-point) method of compute.

We see that the wavefront-gluing technique of HDagg is more effective on the Metis nested dissection reordered data set and also at lower core counts. At higher core counts, in particular on the SuiteSparse data set, it struggles to glue together wavefronts. Under

Data	Cores	HDagg	GrowLocal	ElasticDivide
Suite	**16**	1.34	16.02	6.74
	32	1.15	12.78	5.10
	64	1.06	11.35	4.48
	128	1.07	10.69	4.17
iChol	**16**	1.64	20.74	10.33
	32	1.52	17.35	8.62
	64	1.44	14.58	7.76
	128	1.37	13.30	7.01
Metis	**16**	2.57	16.73	8.59
	32	2.10	14.54	8.15
	64	1.87	12.70	7.22
	128	1.74	11.38	6.73

Table 6.1: Geometric-mean reduction of the number of synchronisation barriers over the number of wavefronts for each algorithm.

these settings, HDagg only becomes marginally better than a wavefront scheduler, which perhaps explains its lack of performance in the higher core count numbers.

Compared to the breadth-first search of wavefront-based schedulers like HDagg, the depth-first search approaches of GrowLocal and ElasticDivide are able to pack in more compute before having to synchronise. Albeit the reported reduction by ElasticDivide is about half the one by GrowLocal, one has to recall that the synchronisations by ElasticDivide are weak and since the staleness is 2, cf. §2.2, its numbers should be multiplied by two. Having taken that into account, we see that ElasticDivide does slightly better than GrowLocal on the Metis data set and slightly worse on the SuiteSparse data set.

6.4 Overlap of Synchronisation and Compute

In order to investigate the effect of overlapping synchronisation and compute, we execute the stale-synchronous-parallel schedule computed by ElasticDivide twice: once using our stale-synchronous-parallel SpTrSV kernel, cf. §4.2, and once using a bulk-synchronous-parallel SpTrSV kernel, implemented using OpenMP [15]. We note that this results in a valid execution, cf. Remark 2.3. In Table 6.2, we display the relative speed-up that the stale-synchronous-parallel SpTrSV kernel achieves over the bulk-synchronous-parallel one.

The data shows a clear picture: overlapping synchronisation and compute is beneficial. In general, we see higher improvements for data sets whose matrices are harder to parallelise such as the SuiteSparse data set and lower numbers for easier to parallelise matrices which are present in the Metis nested dissection data set. We also see that the costlier the synchronisation is, for example through higher non-uniform memory architectures, the larger the improvement is. That is, we see larger speed-ups in ARM than x86 and larger speed-ups for higher core count. There is one exception to the latter and that is the x86 EPYC architecture on the Metis data

Architecture	Cores	Suite	iChol	Metis
ARM Kunpeng	**16**	1.57	1.07	1.06
	32	1.95	1.15	1.13
	48	2.17	1.35	1.21
	64	2.82	1.37	1.34
	80	3.23	1.76	1.48
	96	4.51	2.02	1.65
x86 EPYC	**16**	1.23	1.08	1.07
	32	1.36	1.09	1.07
	48	1.47	1.12	1.06
	64	1.60	1.12	1.05
	80	1.84	1.13	1.03
	96	1.93	1.18	1.01
	112	2.01	1.16	0.96
	128	2.19	1.13	0.99
x86 Xeon	**11**	1.11	1.05	1.03
	22	1.33	1.08	1.12
	33	1.34	1.13	1.12
	44	1.15	1.10	1.08

Table 6.2: Geometric-mean speed-ups of stale-synchronous-parallel execution over bulk-synchronous-parallel execution for all architectures, core counts, and data sets.

set. This phenomenon requires further investigation, though we offer possible explanations: the compiler produces a more efficient type of barrier for this architecture than our barrier, cf. §4.1 and [27], and/or the stale-synchronous-parallel schedule looks kind of like a bulk-synchronous-parallel schedule, that is every other superstep is almost empty.

6.5 Amortisation

An important aspect of the parallel SpTrSV schedules is the time it takes to generates them in the first place and how the quality of the schedule compares to the overhead to generate it in the first place. To this end, we look at how the scheduling time increases with the number of non-zeroes in the matrix. We display the results in Figure 6.2.

We also compute the amortisation costs associated with a schedule [8, 9, 52]. This is defined as the number of executions at which point it is more beneficial to compute a parallel schedule and then execute in parallel, opposed to always using serial execution:

$$n_{\text{amortisation}} = \begin{cases} \frac{t_{\text{schedule}}}{\overline{t_{\text{serial}}} - \overline{t_{\text{parallel}}}}, & \text{if } \overline{t_{\text{serial}}} > \overline{t_{\text{parallel}}}, \\ +\infty, & \text{otherwise,} \end{cases} \tag{6.1}$$

where the bar on top of the times denotes arithmetic mean. In Table 6.3, we record the median amortisation costs for each algorithm,

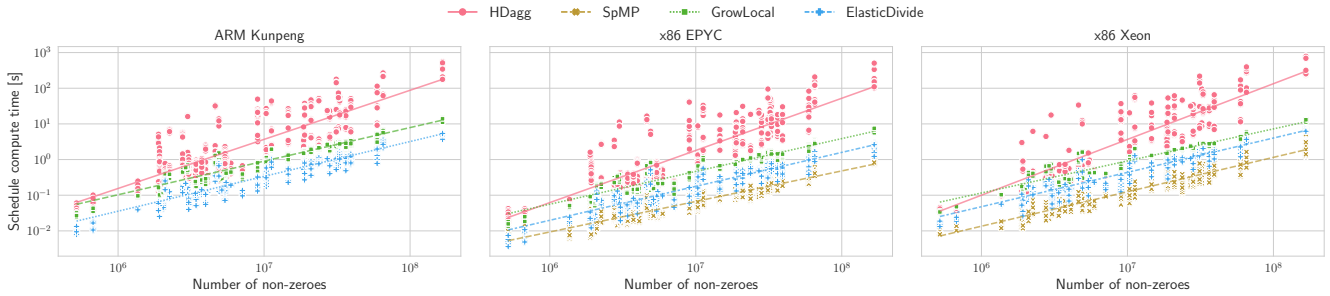


Figure 6.2: Scheduling times of each algorithm plotted against number of non-zeroes of a matrix. The best ℓ^2-fitted line of the shape $\log(y) = m \cdot \log(x) + c$ is also depicted.

Algorithm	Arch.	ARM Kunpeng			x86 EPYC			x86 Xeon		
	Data	Metis	iChol	Suite	Metis	iChol	Suite	Metis	iChol	Suite
HDagg		54.82	211.01	1763.03	85.59	331.20	2819.82	59.52	227.06	1398.14
SpMP		—	—	—	5.41	7.13	9.20	5.88	7.54	7.20
GrowLocal		33.59	35.38	31.10	52.91	53.88	58.28	43.79	46.66	45.51
ElasticDivide		13.60	13.16	11.59	21.32	22.97	17.51	21.60	22.80	25.10

Table 6.3: Median amortisation costs of each algorithm on a full single socket for each architecture and data set.

data set, and architecture, cf. §5. As the core count, we have taken the number of cores corresponding to a single socket.

It is apparent that SpMP is the fastest in scheduling time and that it scales linearly with the number of non-zeroes. This goes to show the engineering efforts that went into the algorithm, in particular the efforts to the make the scheduling algorithm parallel in the first place. ElasticDivide and GrowLocal generate their schedule using a serial algorithm and are therefore slower than SpMP. Nevertheless, their runtime scales (provably) linearly with the number of non-zeroes. HDagg is considerably slower and does not scale linearly. Indeed, the slopes in Figure 6.2 of the best ℓ^2-fitted lines of the shape $\log(y) = m \cdot \log(x) + c$ are between 1.38 and 1.56 for HDagg, whereas they are ≤ 1.0 for all other algorithms. Albeit that ElasticDivide and GrowLocal follow a similar algorithm design, the distance between their ℓ^2-fitted lines in Figure 6.2 indicates that ElasticDivide is twice as fast in computing a schedule. We attribute this to a better implementation with improved data structures. The amortisation cost table, Table 6.3, reflects the scheduling time data depicted in Figure 6.2.

SpMP comes in the lowest with an amortisation cost of 5-10 across x86 architectures. ElasticDivide achieves low amortisation costs, ranging from 12 to 25 across all architectures and data sets. GrowLocal has costs ranging from 31 to 58, performing best on the Metis data set and worst on the SuiteSparse data set. HDagg has the largest amortisation costs, ranging from 55-86 on the most favourable Metis data set all the way up to 2820 on the SuiteSparse data set on x86 EPYC.

7 Conclusion

Synchronising many cores is expensive, more so in non-uniform memory architectures. To combat this expense, one may reduce the number of synchronisations necessary through preprocessing the matrix whenever possible, e.g., nested dissecting, through algorithms that reduce the number of synchronisations to a minimum, and, as we have shown in this paper, through overlapping compute with synchronisation with execution models such as stale-synchronous-parallel. The latter can improve performance easily by 10-30% and when the number of synchronisations is large and the architecture is highly non-uniform by 100% and in some cases up to 350%, cf. §6.4.

Our stale-synchronous-parallel scheduler, ElasticDivide, demonstrates that the fine-grained dependency graphs present in sparse triangular linear systems allow for the necessary flexibility to fit into a stale-synchronous-parallel schedule without (significant) increase in the number of synchronisation events when appropriately normalised. This resulted in geometric-mean speed-ups of around 7-30% against state-of-the-art scheduler GrowLocal on ARM Kunpeng, with even higher speed-ups for higher core and NUMA-domain count, cf. §6.2.

Albeit our presentation and evaluation was conducted on a variety of CPUs, our algorithm ElasticDivide and more generally our ideas and findings are transferable to other compute architectures. For instance, ElasticDivide is directly applicable to solving sparse triangular systems on accelerated compute systems such as GPUs. There it could be used to mask CPU to GPU and cross-GPU latencies with compute.

References

[1] Thomas L. Adam, K. Mani Chandy, and J. R. Dickson. 1974. A comparison of list schedules for parallel processing systems. *Commun. ACM* 17, 12 (1974), 685–690.

[2] Najeeb Ahmad, Buse Yilmaz, and Didem Unat. 2021. A split execution model for sptrsv. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2809–2822.

[3] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. 1996. An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. Appl.* 17, 4 (1996), 886–905. doi:10.1137/S0895479894278952

[4] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* 1, 01 (1989), 73–95.

[5] Cevdet Aykanat, Ali Pinar, and Ümit V. Çatalyürek. 2004. Permuting Sparse Rectangular Matrices into Block-Diagonal Form. *SIAM Journal on Scientific Computing* 25, 6 (2004), 1860–1879. doi:10.1137/S1064827502401953

[6] Rob H. Bisseling, Bas Fagginger Auer, Tristan van Leeuwen, Wouter Meesen, Marco van Oort, Daan Pelt, Brendan Vastenhouw, and Albert-Jan N. Yzelman. 2019. Mondriaan (version 4.2.1). http://www.staff.science.uu.nl/~bisse101/Mondriaan/mondriaan_v4.2.1.tar.gz.

[7] Toni Böhnlein, Benjamin Lozes, Christos K. Matzoros, Pál András Papp, and Raphael S. Steiner. 2024. OneStopParallel. https://github.com/Algebraic-Programming/OneStopParallel.

[8] Toni Böhnlein, Pál András Papp, Raphael S. Steiner, Christos K. Matzoros, and Albert-Jan N. Yzelman. 2025. Efficient Parallel Scheduling for Sparse Triangular Solvers. *arXiv preprint arXiv:2503.05408* (2025).

[9] Toni Böhnlein, Pál András Papp, Raphael S. Steiner, and Albert-Jan N. Yzelman. 2025. Efficient Parallel Scheduling for Sparse Triangular Solvers. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1263–1265. doi:10.1109/IPDPSW66978.2025.00206

[10] Kazem Cheshmi. 2022. *Transforming Sparse Matrix Computations.* Ph. D. Dissertation. University of Toronto, Computer Science. http://hdl.handle.net/1807/123318

[11] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 13, 13 pages. doi:10.1145/3126908.3126936

[12] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric P. Xing. 2013. Solving the Straggler Problem with Bounded Staleness. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. USENIX Association, Santa Ana Pueblo, NM. https://www.usenix.org/conference/hotos13/session/cipar

[13] C. Cleveland Ashcraft, Roger G. Grimes, John G. Lewis, Barry W. Peyton, Horst D. Simon, and Petter E. Bjørstad. 1987. Progress in sparse matrix methods for large linear systems on vector supercomputers. *The International Journal of Supercomputing Applications* 1, 4 (1987), 10–30.

[14] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 37–48. https://www.usenix.org/conference/atc14/technical-sessions/presentation/cui

[15] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[16] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[17] Kiril Dichev, Filip Pawlowski, and Albert-Jan N. Yzelman. 2025. Faster Distributed Inference-Only Recommender Systems via Bounded Lag Synchronous Collectives. *arXiv preprint arXiv:2512.19342* (2025).

[18] Elizabeth D. Dolan and Jorge J. Moré. 2002. Benchmarking optimization software with performance profiles. *Mathematical programming* 91 (2002), 201–213.

[19] Iain S. Duff and Jennifer A. Scott. 2005. Stabilized bordered block diagonal forms for parallel sparse solvers. *Parallel Comput.* 31, 3 (2005), 275–289. doi:10.1016/j.parco.2004.12.008

[20] Xu Fu, Bingbin Zhang, Tengcheng Wang, Wenhao Li, Yuechen Lu, Enxin Yi, Jianqi Zhao, Xiaohan Geng, Fangying Li, Jingwen Zhang, Zhou Jin, and Weifeng Liu. 2023. PanguLU: A Scalable Regular Two-Dimensional Block-Cyclic Sparse Direct Solver on Distributed Heterogeneous Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 51, 14 pages. doi:10.1145/3581784.3607050

[21] Alan George. 1973. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. Numer. Anal.* 10, 2 (1973), 345–363. doi:10.1137/0710032

[22] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[23] Laura Grigori, Erik G. Boman, Simplice Donfack, and Timothy A. Davis. 2010. Hypergraph-Based Unsymmetric Nested Dissection Ordering for Sparse LU Factorization. *SIAM Journal on Scientific Computing* 32, 6 (2010), 3426–3446. doi:10.1137/080720395

[24] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[25] Rajiv Gupta. 1989. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS III)*. Association for Computing Machinery, New York, NY, USA, 54–63. doi:10.1145/70082.68187

[26] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. 2013. More effective distributed ML via a Stale Synchronous Parallel parameter server. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., Red Hook, NY, USA, 1223–1231.

[27] Torsten Hoefler, Torsten Mehlan, Frank Mietke, and Wolfgang Rehm. 2005. *A survey of barrier algorithms for coarse grained supercomputers.* Universitätsbibliothek Chemnitz.

[28] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. 1989. Scheduling precedence graphs in systems with interprocessor communication times. *siam journal on computing* 18, 2 (1989), 244–257.

[29] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[30] Patrick P. C. Lee, Tian Bu, and Girish Chandranmenon. 2009. A lock-free, cache-efficient shared ring buffer for multi-core architectures. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (Princeton, New Jersey) *(ANCS '09)*. Association for Computing Machinery, New York, NY, USA, 78–79. doi:10.1145/1882486.1882508

[31] Xiaoye S. Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.* 31, 3 (Sept. 2005), 302–325. doi:10.1145/1089014.1089017

[32] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. 1979. Generalized Nested Dissection. *SIAM J. Numer. Anal.* 16, 2 (1979), 346–358. doi:10.1137/0716027

[33] Joseph W. H. Liu, Esmond G. Ng, and Barry W. Peyton. 1993. On finding supernodes for sparse matrix computations. *SIAM J. Matrix Anal. Appl.* 14, 1 (1993), 242–252.

[34] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S. Duff, and Brian Vinter. 2016. A synchronization-free algorithm for parallel sparse triangular solves. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22*. Springer, 617–630.

[35] Yang Liu, Nan Ding, Piyush Sao, Samuel Williams, and Xiaoye Sherry Li. 2023. Unified Communication Optimization Strategies for Sparse Triangular Solver on CPU and GPU Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 50, 15 pages. doi:10.1145/3581784.3607092

[36] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient block algorithms for parallel sparse triangular solve. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[37] Jan Mayer. 2009. Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing* 86 (2009), 291–312.

[38] Shang Mingsheng, Sun Shixin, and Wang Qingxian. 2003. An efficient parallel scheduling algorithm of dependent task graphs. In *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 595–598.

[39] Pál András Papp, Georg Anegg, Aikaterini Karanasiou, and Albert-Jan N. Yzelman. 2024. Efficient Multi-Processor Scheduling in Increasingly Realistic Models. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM.

[40] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings 29*. Springer, 124–140.

[41] Andrei Radulescu and Arjan J. C. Van Gemund. 2002. Low-cost task scheduling for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems* 13, 6 (2002), 648–658.

[42] Erik Rigtorp. 2020. SPSCQueue. https://github.com/rigtorp/SPSCQueue.

[43] Edward Rothberg and Anoop Gupta. 1992. Parallel ICCG on a hierarchical memory multiprocessor—addressing the triangular solve bottleneck. *Parallel Comput.* 18, 7 (1992), 719–741.

[44] Joel H. Saltz. 1990. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM journal on scientific and statistical computing* 11, 1 (1990), 123–144.

[45] Joel H. Saltz, Ravi Mirchandaney, and Doug Baxter. 1988. *Run-time parallelization and scheduling of loops.* Technical Report. Institute for Computer Applications in

Science and Engineering, NASA Langley Research Center.

[46] Olaf Schenk and Klaus Gärtner. 2004. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems* 20, 3 (2004), 475–487. Selected numerical algorithms. doi:10.1016/j.future.2003.07.011

[47] Olaf Schenk, Klaus Gärtner, Wolfgang Fichtner, and Andreas Stricker. 2001. PARDISO: a high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Generation Computer Systems* 18, 1 (2001), 69–78. I. High Performance Numerical Methods and Applications. II. Performance Data Mining: Automated Diagnosis, Adaption, and Optimization. doi:10.1016/S0167-739X(00)00076-5

[48] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[49] Leslie G. Valiant. 1990. General purpose parallel architectures. In *Algorithms and Complexity*. Elsevier, 943–971.

[50] Ichitaro Yamazaki, Sivasankaran Rajamanickam, and Nathan Ellingwood. 2020. Performance portable supernode-based sparse triangular solver for manycore architectures. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[51] Buse Yılmaz, Buğrra Sipahioğrlu, Najeeb Ahmad, and Didem Unat. 2020. Adaptive level binning: A new algorithm for solving sparse triangular systems. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. 188–198.

[52] Behrooz Zarebavani, Kazem Cheshmi, Bangtian Liu, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2022. HDagg: hybrid aggregation of loop-carried dependence iterations in sparse matrix computations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1217–1227.

[53] Xing Zhao, Aijun An, Junfeng Liu, and Bao Xin Chen. 2019. Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. IEEE, 1507–1517. doi:10.1109/ICDCS.2019.00150