EFFICIENT PARALLEL SCHEDULING FOR SPARSE TRIANGULAR SOLVERS

TONI BÖHNLEIN[†], PÁL ANDRÁS PAPP[†], RAPHAEL S. STEINER[†],
CHRISTOS K. MATZOROS, AND ALBERT-JAN N. YZELMAN

{toni.boehnlein; pal.andras.papp; raphael.steiner; albertjan.yzelman}@huawei.com
christos.konstantinos.matzoros@h-partners.com

Huawei Research Center Zurich, Computing Systems Lab,
Thurgauerstrasse 80, 8050 Zurich, Switzerland

Abstract. We develop and analyze new scheduling algorithms for solving sparse triangular linear systems (SpTRSV) in parallel. Our approach produces highly efficient synchronous schedules for the forward- and backward-substitution algorithm. Compared to state-of-the-art baselines HDagg [ZCL+22] and SpMP [PSSD14], we achieve a 3.32× and 1.42× geometric-mean speed-up, respectively. We achieve this by obtaining an up to 12.07× geometric-mean reduction in the number of synchronization barriers over HDagg, whilst maintaining a balanced workload, and by applying a matrix reordering step for locality. We show that our improvements are consistent across a variety of input matrices and hardware architectures.

Contents

1. Introduction

Systems of linear equations are ubiquitous and solving them fast numerically with high accuracy is essential to engineering, big data analytics, artificial intelligence, and various scientific fields. Key techniques in scaling to ever larger linear systems have been exploiting the sparsity of non-zero coefficients in modern algorithms, as well as leveraging the multi-core or multi-processor architectures of high-performance computing systems. However, whilst sparsity reduces computational load, the typically irregular distribution of non-zero elements complicates the development of efficient parallel algorithms, as the lack of structure hinders workload balancing and limits the ability to minimize communication between processors.

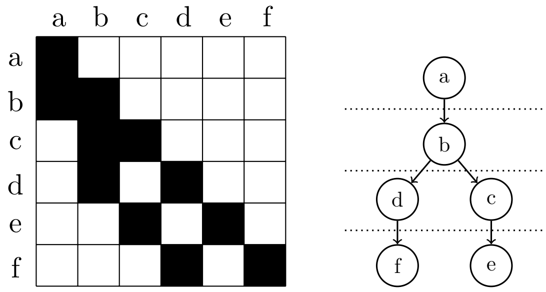
Date: June 8, 2025.

Key words and phrases. Sparse triangular linear system solve, SpTrSV, SpTrSM, forward- and backward-substitution algorithm, barrier list scheduler, synchronous parallel algorithm.

[†] Joint first authors; listed in alphabetical order.

In this paper, we concern ourselves with solving sparse triangular systems of linear equations (SpTRSV) using parallel machines; i.e., solving a linear system $Lx = b$, where L is a sparse triangular matrix and b is a dense vector. Although solving sparse triangular linear systems marks a special case, it often arises as an important step in procedures solving more general linear systems. Some concrete examples are (sparse) LU, QR, and Cholesky decompositions, Gauß–Seidel, and so forth. Efficient parallel-computation schedules for SpTRSV are of particular importance in applications where the same sparsity pattern is used repeatedly. Such is the case in simulations of various physical systems, for instance, ones that are based on the finite element method on a fixed mesh.

One of the main methods of solving SpTRSV is the forward-/backward-substitution algorithm. An execution of the algorithm on an instance may be captured by a directed acyclic graph (DAG), with the vertices corresponding to the rows of the matrix and directed edges representing dependencies imposed by the non-zero entries, see Figure 1.1. Finding a parallel execution of the forward-/backward-substitution algorithm directly corresponds to solving the parallel-scheduling problem on the corresponding DAG.



(A) Sparse lower triangular matrix (6 × 6)

(B) Corresponding DAG

Figure 1.1. A sparse lower triangular matrix (a) and its corresponding DAG for the forward-substitution algorithm (b). Each row of the matrix corresponds to a vertex in the DAG. An edge from vertex u to vertex v exists if and only if there is a non-zero entry in column u of row v in the matrix. The dotted lines in Figure (b) separate the wavefronts of the DAG.

In order to generate an efficient parallel schedule for the algorithm, one needs to:

(i) balance workload across machines, and
(ii) limit coordination overhead.

Satisfying both of these needs simultaneously has proven to be challenging due to the irregular interdependence of computed values and the fine-grained nature of the problem. Early algorithms include so-called wavefront schedulers [AS89, Sal90], which repeatedly schedule all computations whose prerequisites are met, known as the wavefronts, cf. Figure 1.1b, followed by a synchronization barrier. They, however, suffer from large overhead stemming from frequent global synchronization [PSSD14]. Similarly, early asynchronous approaches such as self-scheduling [SMB88] had the drawback of incurring overheads due to numerous fine-grained synchronizations [RG92].

In a breakthrough paper, Park *et al.* [PSSD14] reduced coordination overhead by combining these earlier ideas. Their scheduler SpMP, which remains a competitive baseline to date, is in essence an asynchronous wavefront scheduler: it allows machines to move onto the next wavefront if and only if all requisites have already been met for its portion of the next wavefront. They also developed a fast approximate transitive reduction to reduce the number of synchronization points further. An alternate reduction in synchronizations has been made by Yilmaz *et al.* [YSAU20] by enforcing a bound by which machines may be out of sync.

For synchronous schedulers, efforts have been directed towards increasing the computational load between synchronization barriers, thus decreasing the number of global synchronizations.

For instance, Cheshmi *et al.* [CKSD18] devise such methods for triangular matrices of a special structure, arising in Cholesky decompositions. For general sparse triangular matrices, a state-of-the-art baseline is the recent scheduler HDagg of Zarebavani *et al.* [ZCL+22]. This algorithm develops efficient schedules by gluing together consecutive wavefronts if and only if a balanced workload can still be maintained and by pre-applying a DAG coarsening technique.

1.1. Our contribution. Our work continues along the same path of reducing the number of synchronization barriers. We present and analyze a new scheduling algorithm named *GrowLocal* which is tailored specifically towards the SpTRSV application. In our experiments, we establish that this algorithm produces significantly superior parallel schedules compared to the baseline methods. Specifically, GrowLocal achieves a reduction in execution time of 1.42× compared to SpMP and of 3.32× compared to HDagg, on the SuiteSparse Matrix Collection benchmark [DH11] used by previous studies, see Figure 1.2. We further evaluate GrowLocal on pre-processed variants of the SuiteSparse matrices motivated by applications, and observe speedups of up to 1.80× and 2.20× over SpMP and HDagg, respectively. On synthetic random matrices that are hard to schedule by design, the difference to the baselines is even larger: the algorithm achieves a speed-up of 2.50× compared to SpMP and 10.12× compared to HDagg in execution time.

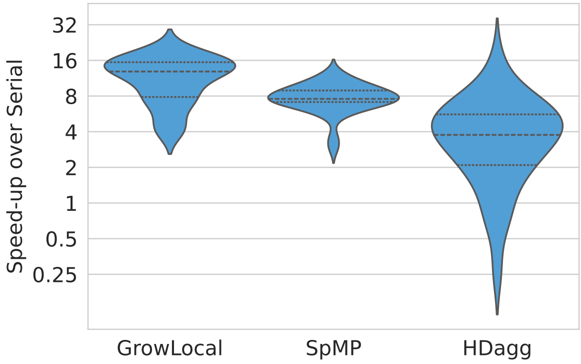


FIGURE 1.2. Geometric mean and interquartile ranges of speed-ups over Serial of our algorithms on the SuiteSparse Matrix Collection [DH11] on an Intel x86 machine using 22 cores.

The algorithm obtains these speed-ups by significantly reducing the number of synchronization barriers required: we report a 12.07× reduction in the number of barriers relative to HDagg on the SuiteSparse data set, whilst maintaining a good workload balance. The results also show that our scheduler provides consistent improvements over several different computing architectures and types of input matrices. The running time of the scheduling algorithm itself is also comparable to the state-of-the-art baselines, making it a viable tool for various applications.

In summary, the main contributions of our paper are:

- a novel algorithm for generating efficient parallel schedules for SpTRSV execution;
- extensions of previous DAG coarsening techniques that enhance the schedules, and a short theoretical proof that these preserve acyclicity; and
- experiments confirming that the above schedulers achieve significant speed-ups over the SpMP and HDagg baselines, on various architectures and data sets, including an ablation study of the individual techniques proposed.

1.1.1. *GrowLocal scheduling algorithm.* There are numerous prior works on parallel DAG scheduling in the literature. When the number of cores is limited, the best results are often achieved by so-called list scheduling algorithms [Gra69, ACD74, HCAL89, RVG02, MSQ03], which schedule the vertices in a topological order according to some priority function [WS18].

On the other hand, DAG scheduling with barrier synchronization is a somewhat different setting, and there are only a few previous works that address this problem. One state-of-the-art example here is the HDagg algorithm mentioned before [ZCL+22], which can also be interpreted as scheduler for general DAGs. Besides this, the idea of adapting list schedulers to a barrier synchronization setting has also been explored recently by Papp *et al.* [PAKY24] for abstract bulk-synchronous-parallel (BSP) scheduling.

Our GrowLocal algorithm takes a rather different approach than these previous methods, but also incorporates some of their underlying strengths. On a high level, GrowLocal considers a parameter α, and tries to form the part of the schedule until the next synchronization barrier (the next so-called *superstep*) by assigning approximately α vertices to each of the cores before this next barrier. The parameter α is then iteratively increased, examining larger parts of the DAG as the potential next superstep, as long as this is possible while also ensuring a sufficient amount of parallelization between the cores.

During the development of the schedule, the algorithm always maintains the set of vertices that are ready to be executed, i.e., all their parents have been computed. At any point during the algorithm, if we consider the current superstep, such a ready vertex v may be executable either on any of the cores (if all parents of v were computed before the last barrier), only on a specific core p (if a parent of v was computed on p since the last barrier), or on none of the cores (if we computed parents of v on multiple cores since the last barrier). When selecting the next vertices to assign to a core p during our algorithm, GrowLocal first prioritizes vertices that can only be executed on the core p before the next barrier. This is inspired by the scheduler of [PAKY24], and it ensures that we can compute significantly more vertices before having to insert a new barrier.

Apart from this, GrowLocal simply selects vertices to assign to a core based on their IDs in order to group neighboring vertices onto the same core and superstep. This leads to significantly better locality for the developed schedule than in case of, e.g., list schedulers, and this has a large positive impact on the overall performance of the SpTRSV computation.

1.1.2. *Coarsening.* Graph coarsening techniques are widely applied in graph partitioning tools [IKS75, KAKS97, Sch20], where they greatly reduce the size of the graph and improve data locality. These techniques can also be applied to DAG scheduling [PSSS21, ZCL+22], where they can further help to reduce the number of synchronization steps on top of the aforementioned benefits. Following the coarsening, the scheduling algorithm is applied to the coarse graph and the resulting schedule is subsequently pulled back to the original graph to obtain the final schedule. In order to produce a valid scheduling problem, the coarsening methods are required to preserve the acyclicity of DAGs. Methods that fulfill this criteria have been studied in several works before, see, for example, [CLB94, FER+13, HKU+17, ZCL+22] and references therein.

In Section 4, we introduce the concept of *cascades* to generalize the coarsening techniques utilized in [CLB94, §4] and [ZCL+22, §IV.B]. We then formally prove that coarsening techniques based on cascades always preserve acyclicity. In Section 7.3, we evaluate the effect of the coarsening algorithm developed in Section 4 on our scheduling algorithm, GrowLocal.

1.1.3. *Reordering.* Besides the algorithms above, we also apply a matrix reordering step to drastically improve data locality during the SpTRSV computation. Specifically, once the schedule is developed, we symmetrically permute the matrix according to the schedule, ensuring that values computed after each other on the same core are close to each other in this permuted representation. This idea has already been explored by Rothberg–Gupta in the 1990s [RG92], but it has not been applied in modern SpTRSV baselines, which instead try to make use of existing data locality when deriving a schedule.

1.1.4. *Block parallel scheduling.* A known optimization technique for parallel SpTRSV execution is to break the lower triangular matrix into blocks [AS89, May09, AYU21, YSAU20]. These blocks may be on the diagonal, which corresponds to a smaller instance of (sparse) triangular solve, or completely off the diagonal, which corresponds to a (sparse) matrix-vector multiplication. The separation of the easily parallelizable (sparse) matrix-vector-multiplication blocks from the hard to parallize (sparse) triangular blocks has been particularly impactful for GPU implementations [LLH$^+$16, LNL20].

In this paper, we use this block decomposition to run the GrowLocal scheduling algorithm in parallel on each (sparse) triangular block. The resulting synchronous schedules can then be combined one after the other (with a synchronization barrier between individual schedules) to a schedule for the whole triangular matrix. This leads to a super-linear speed-up in the scheduling time whilst having a moderate effect on the parallel SpTRSV solve time.

1.2. Additional related work. Besides the forward-/backward-substitution algorithm, there are also other methods for solving sparse triangular systems, for example inversion. For this method, we mention the memory-optimal algorithms developed in previous works [AS93, PA92].

2. Background

2.1. Graph notation. We model our computations as a directed acyclic graph (DAG) $G = (V, E)$, which consists of a set of vertices V and a set of directed edges $E \subseteq V \times V$. For any vertex $v \in V$, the sets of vertices $\{u \,|\, (u, v) \in E\}$ and $\{u \,|\, (v, u) \in E\}$ are called the *parents* of v and the *children* of v, respectively. The *in-* and *out-degree* of v, denoted by $\deg^-(v)$ and $\deg^+(v)$, respectively, are the number of parents and children of v. The *degree* of v, denoted by $\deg(v)$, is the sum of its in- and out-degree. If a vertex of the DAG has no parents/children, then it is called a *source/sink* vertex, respectively. The DAG in our model is also complemented by vertex weights $\omega : V \to \mathbb{Z}_{>0}$ to indicate the compute cost of each operation.

2.2. Problem definition and notation. When solving sparse triangular systems, we are given a triangular matrix $A = (A_{i,j})_{i,j=1,\ldots,n} \in \mathbb{R}^{n\times n}$, a dense vector $b = (b_1, \ldots, b_n)^T \in \mathbb{R}^n$, and the goal is to solve the equation $Ax = b$ for the vector $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$. We assume that A is non-singular, such that all its diagonal elements are non-zero. In case of a lower triangular matrix A, there is a natural *forward-substitution algorithm* for the problem, which iterates through the rows of A in order and computes the values of x as $x_1 = \frac{b_1}{A_{1,1}}$, $x_2 = \frac{b_2 - A_{2,1}x_1}{A_{2,2}}$, and, in general, as

$$x_i = \frac{1}{A_{i,i}} \left(b_i - \sum_{j=1}^{i-1} A_{i,j} x_j \right). \tag{2.1}$$

In case of an upper triangular matrix A, a backward-substitution algorithm follows symmetrically in the reverse direction.

In the forward-substitution algorithm (2.1), we say that the computation of x_i *depends* on the value of x_j, for $j < i$, if and only if there is an increasing sequence $j = \ell_0 < \ell_1 < \cdots < \ell_m = i$ such that each entry A_{ℓ_{k-1},ℓ_k} is non-zero, for $k = 1, \ldots, m$. If there is no dependency between x_i and x_j, the two corresponding operations can be executed in any order, in particular also in parallel. As such, the operations in the algorithm can naturally be represented as a DAG $G = (V, E)$, where $V = \{1, \ldots, n\}$, the vertex i represents the i-th row of A, and, for any $i, j \in V$, we have a directed edge $(j, i) \in E$ if and only if $A_{i,j} \neq 0$. See Figure 1.1 for an example. To indicate the compute cost of each operation, the weight $\omega(v)$ of each vertex $v \in V$ in the DAG is simply defined as the number of non-zero entries in the corresponding row of the matrix.

The parallel execution of this DAG then directly corresponds to a parallel execution of the SpTRSV. Many previous works found it more convenient to discuss their scheduling methods for this problem using this DAG representation.

The parallel-scheduling problem above can be most fittingly captured in a bulk-synchronous parallel (BSP) model [Val90a] that assumes *global synchronization barriers* to split the execution into so-called *supersteps*. This model is also known as the XPRAM model [Val90b]. A schedule in this model assigns each vertex, i.e., the computation of each x_i, to one of the k available cores and to a given superstep. A valid schedule must fulfill the precedence constraints of the DAG and ensure that we always have a synchronization barrier between computing a value on one core and using it as input on another core.

Definition 2.1. *A* parallel schedule *of G consist of assignments $\pi : V \to \{1, ..., k\}$ to cores and $\sigma : V \to \mathbb{Z}_{>0}$ to supersteps, which fulfill the following properties for each $(u, v) \in E$:*

- *$\sigma(u) \leqslant \sigma(v)$;*
- *if $\pi(u) \neq \pi(v)$, then $\sigma(u) < \sigma(v)$.*

The total cost of a schedule is determined by the workload balance within each superstep and the number of synchronization barriers. The original BSP model includes also communication volume in its cost function. For the SpTRSV application, however, the communication happens in parallel to the computation and resolving the synchronizations. Hence, the latter two dominate the overall execution time. Synchronous methods from previous works apply the same scheduling model, although often without explicitly referring to BSP, XPRAM, or supersteps.

3. The GrowLocal scheduler

Our GrowLocal algorithm is tailored specifically to the DAG scheduling problem with synchronization barriers. The algorithm forms the supersteps one by one, always aiming to make the current superstep as large as possible while maintaining a good workload balance. The current superstep is formed through several *iterations* with a superstep length parameter α. The algorithm attempts to form a new superstep with approximately α vertices assigned to each core, and gradually increases α as long as this allows sufficient parallelization.

Specifically, in a single iteration with parameter α, the algorithm first assigns (up to) α vertices to the first core, and considers the sum Ω_1 of the weights of these vertices. It then assigns vertices up to total weight of at most Ω_1 to the second core, third core, and so forth. Let Ω_p denote the total weight allocated to core p in this iteration. We associate a parallelization score of

$$\beta = \frac{\sum_p \Omega_p}{\max_p \Omega_p + L} \tag{3.1}$$

to the current iteration. Here, L is a parameter reflecting the penalty (time cost) incurred by each new synchronization barrier[1]. In order to consider the superstep allocation of the current iteration *worthy*, the algorithm requires that its score β is relatively large, i.e., close to the parallelization score achieved in the previous iterations.

In order to form a superstep, the algorithm begins with a minimal length $\alpha = 20$ iteration. This first iteration is always considered worthy, regardless of its parallelization score. Then, in each subsequent iteration, we consider a different choice for the next superstep: the assignments of the previous iteration are invalidated, the parameter α is increased by a factor of 1.5, and a new potential superstep is formed, with more vertices assigned to each core. If the resulting parallelization score is still high enough, then the superstep allocation of this iteration is also considered worthy, and the process continues. Otherwise, the last worthy superstep allocation is finalized as the current superstep. The high-level pseudocode of the algorithm is outlined in Algorithm 3.1.

Naturally, when assigning vertices to a specific core p in a superstep, there may be numerous ready-to-compute vertices that we can choose from, and selecting among these is a key aspect to any scheduler. Similarly to the heuristic of [PAKY24], our algorithm first prioritizes those vertices that are only computable on p in this superstep, since some of their parents were assigned to p in the current superstep. In lack of such vertices, GrowLocal simply selects the vertices with smallest IDs.

[1] The value of L may be architecture dependent. In this study, we set $L = 500$ based on synchronization cycles and a small empirical evaluation.

Algorithm 3.1: Skeleton of GrowLocal scheduler

```
Data: A vertex-weighted DAG $G = (V, E, \omega)$ and a set of cores $P = \{1, 2, \ldots, k\}$.
Result: A schedule consisting of processor assignment $\pi : V \to P$ and superstep
        assignment $\sigma : V \to \mathbb{Z}_{>0}$.
Rule I: Vertices are prioritized according to
          (i) core exclusivity, and then
         (ii) smallest ID.

while not all vertices are assigned yet do
    $\alpha \leftarrow 20$
    while true do
        // I. Assign new vertices to each core
        assign up to $\alpha$ vertices to core 1 with Rule I
        $\Omega_1 \leftarrow$ total newly assigned weight to core 1
        for core $p = 2, \ldots, k$ in order do
            $\Omega_p \leftarrow 0$
            while $\Omega_p \not\approx \Omega_1$ and can assign to core $p$ do
                assign vertex $v$ to core $p$ with Rule I
                $\Omega_p \leftarrow \Omega_p + \omega(v)$
        // II. Check for sufficient parallelism
        $\beta \leftarrow \frac{\sum_p \Omega_p}{\max_p \Omega_p + L}$
        if the parallelization score $\beta$ is high enough then
            consider current assignment as worthy
            undo new assignments up to the last barrier
            $\alpha \leftarrow 1.5 \times \alpha$
        else
            finalize last worthy assignment as next superstep
            break inner loop
```

We note that while this ID-based selection may seem simple at first, it in fact plays a crucial role in the success of our algorithm. Previous scheduling heuristics usually assign vertices to the different cores simultaneously in order to ensure work balance. In contrast to this, GrowLocal first assigns vertices to the first core, then to the second core, and so forth. With the ID-based selection, this often leads to schedules where the vertices on a core are more-or-less consecutive blocks in the matrix, which drastically improves locality during the computation. This is especially important in matrices from applications, which are often already ordered superbly with respect to locality, and thus preserving this is crucial. As such, GrowLocal can also be loosely understood as a method combining the strengths of previous DAG schedulers with barrier synchronization: similarly to [PAKY24], it allows a priority-based choice between vertices when forming a superstep, but as in [ZCL+22], it preserves locality by aiming to assign consecutive vertices to the same core.

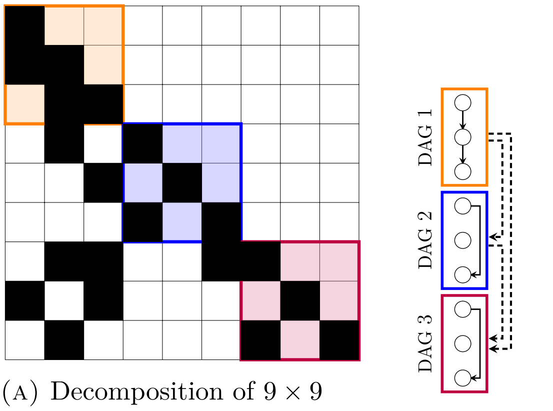
Under reasonably mild assumptions, one can also show that the running time of the algorithm is almost linear.

Theorem 3.1. *Assume that both the out-degrees and compute weights of the DAG are on the same order of magnitude. Then, the time complexity of the GrowLocal algorithm is* $O(|E| \cdot \log |V|)$.

The precise formulation of the theorem and the proof are deferred to Section B of the supplementary material. On a high level, since the size of the iterations follows a geometric series, one can show the total number of speculative vertex assignments in a superstep is still only a linear factor more than the size of the finalized superstep. However, a rigorous proof of the theorem is much more technical due to the fact that our parallelization score also depends on the weights of the vertices and the parameter L. In the supplementary material, we also provide a brief experimental analysis, which confirms linear complexity.

One can also easily observe that the space requirement of the algorithm is simply $O(|E|)$.

3.1. Block parallel scheduling. In order to reduce the overhead from scheduling even further, one can parallelize the scheduling process. Instead of directly parallelizing the algorithm, we split the scheduling problem into independent parts. We achieve this by subdividing the lower triangular matrix into smaller lower triangular matrix blocks along the diagonal as in Figure 3.1.



(A) Decomposition of 9 × 9 matrix

(B) Sub-DAGs

FIGURE 3.1. Subdivision of a 9 × 9 lower triangular matrix into three 3 × 3 lower triangular matrix blocks (a) and the three corresponding sub-DAGs with inter-DAG dependency (b).

On each of these sub-problems, we can generate schedules in parallel. When combining these schedules, we just have to ensure that the individual schedules are combined one after the other. Equivalently, we can add to the superstep assignment of each vertex in each block the total number of supersteps in earlier block schedules.

We remark that for the weight of the vertices (in the DAG representation), we still use the number of non-zeros in the full matrix. This is in line with our SpTRSV kernel implementation.

4. Acyclicity-preserving graph coarsening

Previous works discuss several ways to partition a DAG into clusters such that this coarsened graph remains acyclic, although often without a formal proof of this property. In a further generalization of earlier methods from Cong *et al.* [CLB94, §4] and Zarebavani *et al.* [ZCL$^+$22, §IV.B], we now introduce the concept of *cascades* and prove that coarsening a DAG along such cascades is still guaranteed to preserve acyclicity. This is presented in Section 4.1. In Section 4.2, we describe the graph coarsening algorithm used in our scheduling algorithms.

4.1. Cascades. We begin with some formal definitions. Thereafter, we prove Proposition 4.3, demonstrating the utility of cascades for coarsening DAGs.

Definition 4.1. *Let $G = (V, E)$ be a directed graph and P a partition of V. We define the* coarsened graph *of G along P as the graph (V', E'), where $V' = P$, i.e., the vertices are the parts of the partition P, and for $U', W' \in V'$ we have that $(U', W') \in E'$ if and only if $U' \neq W'$ and $\exists(u, w) \in E$ such that $u \in U'$ and $w \in W'$. We denote the coarsened graph of G along P by $G/\!/P$.*

In other words, the coarsened graph $G/\!\!/P$ is the graph G quotiented by the equivalence relation induced by P with self-loops removed. The definition is easily extended to vertex-weighted graphs, where the weight of a part $U \in P$ is given as the sum the weights of its elements: $\omega(U) = \sum_{u\in U} \omega(u)$.

Definition 4.2. *Let $G = (V, E)$ be a directed graph. We call a subset of vertices $U \subseteq V$ a* cascade *if and only if for every vertex $v \in U$ with an incoming cut edge, that is $(w, v) \in E$ such that $w \notin U$, and for every vertex $u \in U$ with an outgoing cut edge, that is $(u, w) \in E$ such that $w \notin U$, there is a (possibly trivial) directed walk from v to u in G.*

Proposition 4.3. *Let $G = (V, E)$ be a directed acyclic graph and P a partition of V such that each set $U \in P$ is a cascade. Then, the coarsened graph $G/\!\!/P$ of G along P is acyclic.*

Proof. We will show that any directed walk in $G/\!\!/P$ can be elevated to a directed walk in G. Therefore, the existence of directed cycles in $G/\!\!/P$ implies the existence of directed cycles in G.

We lift a walk from $G/\!\!/P$ by mapping each edge to an arbitrary representative in E, whose endpoints necessarily lie in disjoint sets of the partition P as $G/\!\!/P$ does not contain any self-loops, and connecting the endpoints via the directed walks guaranteed by the defining property of cascades. □

4.2. Algorithm. In our graph coarsening algorithm, we do not make use of the full strength of Proposition 4.3. Instead, we use a subcategory of cascades, which can be found efficiently. We call them *funnels*, though they have been previously described under the name *fanout-free cone* [CLB94, §4]. Since the latter reference does not include an algorithm with a complexity analysis, we include them here in Algorithm 4.1.

Definition 4.4. *Let $G = (V, E)$ be a directed acyclic graph. We call a subset of vertices $U \subseteq V$ an* in-funnel *if and only if U is a cascade and there is at most one vertex $u \in U$ with an outgoing cut edge, that is $(u, w) \in E$ such that $w \notin U$.*

We analogously define an out-funnel.

The time complexity of the topological sort is $O(|V| + |E|)$ [Kah62] and its space complexity is $O(|V|)$. In order to bound the time complexity for the remaining part, we note that each parent vertex v in Line 11 is visited at most as many times as its out-degree, leading to an overall complexity of $O(|V| + |E|)$. The space complexity is easily seen to be $O(|V|)$.

In practice, before applying this graph coarsening, we remove some transitive edges from G as this increases the likelihood of finding larger components. A complete transitive reduction is slow, though there are faster approximate transitive reductions, such as the 'remove all long edges in triangles'-algorithm [PSSD14, §2.3] with a time complexity of $O(\sum_{v\in V} \deg(v)^2)$. This algorithm may be terminated early if a faster runtime is desired. In our implementation, we run the full (approximate) algorithm.

In our implementation, we also add a size/weight constraint on each component of the partition to Algorithm 4.1 as otherwise a graph with only one sink vertex would be coarsened into a graph with only one vertex. Altogether, our coarsening algorithm Funnel is a more general[2] and robust version of the coarsening algorithm in HDagg [ZCL+22].

[2] Every *in-tree* is an *in-funnel*.

Algorithm 4.1: In-funnel graph coarsening.

```
Data: A directed acyclic graph $G = (V, E)$.
Result: A partition $P$ such that every $U \in P$ is an in-funnel.
Partition $\leftarrow \emptyset$
Visited$[v] \leftarrow$ false, $\forall v \in V$
for $v \in V$ in reverse topological order do
    if Visited$[v]$ then continue
    $U \leftarrow \emptyset$
    ChildrenCount$[u] \leftarrow 0$, $\forall u \in V$
    PrioQueue.insert$(v)$
    while not PrioQueue.empty() do
        $w \leftarrow$ PrioQueue.pop()
        $U$.insert$(w)$
        for $u \in$ Parents$(w)$ do
            ChildrenCount$[u] \leftarrow$ ChildrenCount$[u] + 1$
            if ChildrenCount$[u] =$ OutDegree$(u)$ then
                PrioQueue.insert$(u)$
    for $u \in U$ do
        Visited$[u] \leftarrow$ true
    Partition.insert$(U)$
return Partition
```

5. Reordering for locality

Our algorithms already account for two of the most important factors in synchronous scheduling: work balance and the number of synchronization barriers. However, another major aspect that greatly influences the efficiency of a parallel SpTRSV execution is data locality, i.e., the number of required values that are already available in cache.

In order to address this, we apply a separate reordering step to ensure that vertices which are computed together are also stored together. The main idea of this approach has already been considered before, cf. [RG92], but has not found its way into modern baselines. In particular, we consider a reordering (relabeling) the vertices of the input DAG based on the partitioning we developed, where we iterate through the supersteps in order, and within each superstep, we iterate through the cores in order. That is, we first start with the vertices v with $\pi(v) = 1$, $\sigma(v) = 1$, then the vertices v with $\pi(v) = 2$, $\sigma(v) = 1$, and so on, up to the vertices v with $\pi(v) = k$, $\sigma(v) = 1$, followed by vertices v with $\pi(v) = 1$, $\sigma(v) = 2$, etc. Within a given core-superstep combination, we go through the vertices in the original order (which gives a topological ordering of the induced sub-DAG). We then symmetrically permute the input matrix and permute the right-hand-side vector of the SpTRSV problem accordingly. Note that since the permutation provides a valid topological ordering of the vertices of the DAG, the resulting matrix is still lower triangular, resulting in an equivalent (symmetrically permuted) formulation of the SpTRSV problem.

We then execute the SpTRSV computation on the permuted problem, following our schedule, which ensures that vertices computed on the same core in the same superstep are stored close to each other, thus greatly improving locality during the computation.

6. Experimental setup

In this section, we present the experimental setup for the evaluation of our scheduling algorithm. Our implementations are available in the *OneStopParallel* repository [BLM^{+}24] on Github.

6.1. Methodology. For the evaluation, we used a standard SpTRSV implementation which iterates through the rows of the matrix which was stored in compressed sparse row (CSR) format [TW67]. The algorithm was parallelized using the OpenMP library with the flags `OMP_PROC_BIND` and `OMP_PLACES` set to `close` and `cores`, respectively.

We measured one hundred times the time it takes for a single SpTRSV execution using the chrono high-resolution clock. The measurements were taken whilst the system was 'hot', meaning two untimed executions precede the timed executions. Between each SpTRSV execution, the right-hand-side vector b was reset to all ones. The experiments were repeated for each scheduling algorithm, data set, and CPU architecture type. The latter two are described in more detail in Section 6.2 and Section 6.3, respectively. If the interquartile range of the measurements corresponding to a scheduling method was too large, we rejected and re-ran all experiments on the same matrix and processor configuration.

The experiments for the schedulers HDagg and SpMP were carried out in the sympiler framework [CKSD17, Che22] as in [ZCL$^+$22] with only minor adjustments to adhere to the aforementioned setup. All remaining schedulers were tested in our own framework.

All scheduling algorithms are implemented in C++ and were compiled with GCC (11.4.0 or 11.5.0) using the optimization flag `-O3`.

6.2. Data sets. For the experiments, we used matrices from several data sets. The main data set is a sample from the SuiteSparse Matrix Collection [DH11], which constitutes a diverse set of matrices from a wide range of applications and was used in previous studies [ZCL$^+$22]. We also consider two modified versions of this data set that are also relevant in their own right. Finally, these data sets are complemented with two randomly generated ones: uniformly random, i.e., Erdős–Rényi matrices [ER59], and random with a bias towards the diagonal. The former are easier to parallelize as they have few (and thus large) wavefronts [HKSL14] and the latter are specifically designed to be harder to parallelize, though they admit good locality.

A useful general metric to understand the parallelizability of an SpTRSV execution is the average wavefront size, which can be calculated from the DAG representation by dividing the number of vertices by the length of the longest path. This metric is indicated for each matrix in the overview of the data sets in the supplement, see Section A.

6.2.1. *SuiteSparse.* From the SuiteSparse Matrix Collection [DH11], we used the lower triangular part of all the sparse real symmetric positive definite matrices. Out of those, we further restricted ourselves to large matrices with enough available parallelism, meaning

- the number of floating point operations[3] is at least 2 million, and
- the average wavefront size is at least 44, twice the number of cores utilized in the experiments.

We furthermore removed matrices from the data set which had the same sparsity pattern. An overview over some statistics of the matrices may be found in Table A.1 of the supplement.

6.2.2. *SuiteSparse METIS (METIS).* Zarebavani *et al.* [ZCL$^+$22] also use the real symmetric positive definite matrices from the SuiteSparse Matrix Collection as their data set. However, they use a modified version of this data set, which we also reproduce here. In their experiments, the matrices are first symmetrically permuted using a fill-reducing method of METIS [KK98] and only then the lower triangular part is taken. In general, this results in non-equivalent SpTRSV problems. The sparsity pattern of 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. An overview over some statistics of the matrices may be found in Table A.2 of the supplement.

[3] The number of floating point operations is equal to twice the number of non-zeros minus the dimension of the matrix.

6.2.3. *SuiteSparse Eigen incomplete Cholesky (iChol).* This data set consists of lower triangular matrices obtained after an incomplete Cholesky decomposition. The initial set of matrices are the same symmetric matrices used in the SuiteSparse data set[4]. The incomplete Cholesky decomposition was performed using the 'IncompleteCholesky' method of Eigen [GJ+10] using the built-in fill-reducing method 'AMDOrdering'. An overview over some statistics of the matrices may be found in Table A.3 of the supplement.

6.2.4. *Erdős–Rényi.* These are lower triangular matrices where each entry (i, j), with $i > j$, is independently non-zero with a fixed probability p. The values of the non-zero non-diagonal entries we have chosen to be independently uniformly distributed in $[-2, 2]$. The absolute value of the diagonal entries we have chosen to be independently log-uniformly distributed in $[2^{-1}, 2]$ and their sign to be ± independently uniformly random[5]. The DAGs corresponding to these matrices are directed Erdős–Rényi random graphs [ER59].

We generated thirty $N \times N$ matrices of this type with $N = 100{,}000$ and $p = 10^{-4}, 5 \cdot 10^{-4}, 2 \cdot 10^{-3}$, ten of each given probability. An overview over some statistics of the matrices may be found in Table A.4 of the supplement.

6.2.5. *Narrow bandwidth.* We also create a data set of random matrices which are much harder to parallelize by design, but have good locality. Unlike the Erdős–Rényi random matrices, we let the lower triangular matrix entry (i, j), with $i > j$, be independently non-zero with probability $p \cdot \exp((1 + j - i)/B)$, moving the non-zero entries closer to the diagonal. The entry values were chosen as in Section 6.2.4.

We generated thirty $N \times N$ matrices of this type with $N = 100{,}000$ and $(p, B) = (0.14, 10)$, $(0.05, 20)$, $(0.03, 42)$, ten for each pair (p, B). An overview over some statistics of the matrices may be found in Table A.5 of the supplement.

6.3. CPU architectures. The CPU architectures used for the experiments were x86 and ARM. The precise model and some specifications are given, respectively, as follows:

- Intel Xeon Gold 6238T processor (x86), with 192 GB memory and theoretical peak memory throughput of 140.8 GB/s and 22 cores on a single socket; kernel version 5.14.0; GCC version 11.5.0;
- AMD EPYC 7763 processor (x86), with 1024 GB memory and theoretical peak memory throughput of 204.8 GB/s and 64 cores on a single socket; kernel version 5.15.0; GCC version 11.4.0;
- Huawei Kunpeng 920-4826 (Hi1620) processor (ARM), with 512 GB memory and theoretical peak memory throughput of 187.7 GB/s and 48 cores on a single socket; kernel version 5.15.0; GCC version 11.4.0.

7. Evaluation

7.1. Overall performance. We present speed-ups of the forward-/backward-substitution algorithm based on parallel schedules compared to serial execution. The schedules of our proposed algorithm are benchmarked against those produced by the baseline methods, SpMP [PSSD14] and HDagg [ZCL+22]. The results, aggregated over the instances from the respective data set using the geometric mean of all pairs of runs, are displayed in Table 7.1. All experiments were conducted on the Intel x86 machine utilizing 22 cores.

On our main data set, SuiteSparse, the schedules generated by our GrowLocal algorithm achieves a geometric-mean speed-up of 1.42× compared to SpMP and 3.32× compared to HDagg. We also see similar results on the two variations of the SparseSuite data set: on METIS, GrowLocal obtains a 1.70× and 1.77× geometric-mean speed-up to SpMP and HDagg, respectively, and on iChol, it achieves a 1.80× and 2.20× speed-up to SpMP and HDagg, respectively. This shows that GrowLocal indeed significantly outperforms the baseline algorithms on these application-based data sets.

[4] The matrix 'bundle_adj' segmentation-faults during the process and is thus excluded from the data set.

[5] The change of distribution on the diagonal is to avoid numerical instability, in particular divisions by zero.

The differences are even larger on the Narrow Bandwidth matrices: here GrowLocal achieves a 2.50× and 10.12× factor improvement compared to SpMP and HDagg, respectively. This indicates that in the more challenging cases when our DAGs are particularly hard to parallelize, GrowLocal is even more superior to the baselines.

Finally, on the Erdős–Rényi data set, the improvement is much smaller; since these DAGs are easier to parallelize, the differences between the algorithms become less relevant.

We also note that Funnel coarsening does not seem to further improve GrowLocal; we elaborate on this later in Section 7.3.

Data set	GrowLocal	Funnel+GL	SpMP	HDagg
SuiteSparse	**10.79**	10.19	7.60	3.25
METIS	**15.93**	15.40	9.35	9.00
iChol	**15.10**	14.84	8.36	6.87
Erdős–Rényi	**12.75**	12.66	9.38	8.44
Narr. bandw.	**9.04**	8.26	3.56	0.88

Table 7.1. Geometric mean of speed-ups over serial execution of GrowLocal with/without Funnel coarsening, compared to the baselines SpMP and HDagg on the Intel x86 machine using 22 cores taken over the data sets from Section 6.2.

We also include a performance profile [DM02] based on the data generated from the SuiteSparse data set in Figure 7.1. The closer the line is to the top left corner, the better and more consistent the algorithm is across the data set. This shows that our algorithm is not only faster in execution time on average but it is so throughout the diverse SuiteSparse data set.

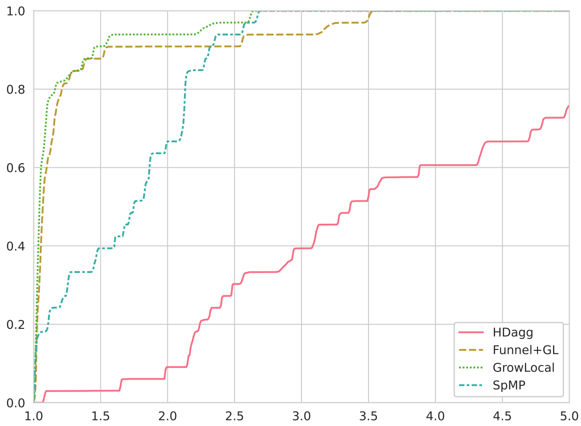


Figure 7.1. Performance profiles of our algorithms on the SuiteSparse data set evaluated on the Intel x86 machine using 22 cores. The x-axis represents a threshold and the y-axis is the proportion of runs that are within this threshold times the fastest SpTRSV run on the respective matrix.

7.2. Fewer synchronization barriers. The results in Table 7.1 show that our scheduler can significantly outperform the synchronous state-of-the-art HDagg. A further analysis shows that this is in part due to a substantial reduction in the number of synchronization barriers required during execution, whilst still maintaining a good work balance. In particular, Table 7.2 shows the number of synchronization barriers relative to the number of wavefronts in our algorithm and HDagg. The data indicates a large, up to 14.99×, reduction of number of synchronization barriers compared to the number of wavefronts on the SuiteSparse data set. This is a reduction of up to 12.07× compared to HDagg, which explains the significant speed-ups achieved by our methods. In general, we can observe a similar effect in the remaining data sets, but the difference is much smaller for Erdős–Rényi, and much higher for narrow bandwidth matrices.

Data set	**GrowLocal**	**Funnel+GL**	**HDagg**
SuiteSparse	14.99	**17.09**	1.24
METIS	16.55	**21.83**	2.39
iChol	18.91	**22.86**	1.62
Erdős–Rényi	2.93	**2.99**	1.25
Narrow bandw.	**51.12**	42.00	1.10

Table 7.2. Geometric mean of the reduction of the number of synchronization barriers relative to the number of wavefronts of the matrix within each data set from Section 6.2.

7.3. Impact of Funnel coarsening. The results in Table 7.1 show that the DAG coarsening approach does not allow to further improve the schedules developed by GrowLocal on most of the data sets. This is an interesting contrast to the HDagg baseline, where coarsening is also a key ingredient of the scheduler. This suggests that GrowLocal is already rather strong at exploring the DAG structure and exploiting locality, and hence, the advantages of the coarsening step cannot compensate for the loss of structure in the graph.

However, besides this negative result, the Funnel coarsener also has several benefits that make it interesting in its own right. Firstly, it allows to run GrowLocal on a much smaller DAG, and as a results, the combined running time of Funnel+GrowLocal is often lower than GrowLocal alone; we will quantify this later in Section 7.7. As such, Funnel+GrowLocal can be a more desirable alternative when the scheduling time is also a critical factor. Secondly, Funnel coarsening allows one to reduce the number of synchronization barriers even further: while GrowLocal achieves a 12.07× geo-mean reduction compared to HDagg, Funnel+GrowLocal together achieves a 13.76× geo-mean reduction of synchronization barriers, which is of independent interest.

7.4. Impact of reordering. We separately analyze the impact of the reordering step on the performance. Table 7.3 compares the speed-ups achieved by our algorithm with and without the reordering component from Section 5. The numbers show that reordering is indeed a valuable ingredient of our scheduler. The data also confirms that even without the reordering, the algorithm still outperforms HDagg notably, which is the current state-of-the-art synchronous baseline, cf. Table 7.1.

Data set	**Reordering**	**No Reordering**
SuiteSparse	10.79	8.62
METIS	15.93	15.21
iChol	15.10	15.02
Erdős–Rényi	12.75	7.87
Narrow bandw.	9.04	6.96

Table 7.3. Geometric mean of speed-ups relative to Serial of GrowLocal with/without permuting the matrix data according to the computed schedule. Experiments were conducted on the Intel x86 machine using 22 cores.

7.5. Performance across different architectures. We show the performance gains of our algorithm over the different processors and architectures in Table 7.4. The data confirms that our algorithm consistently outperforms the baselines across all considered architectures. We note that the improvement relative to Serial can be in a significantly different range due to the properties of the distinct architectures. SpMP is omitted for the ARM architecture because its implementation is x86-specific.

Machine	GrowLocal	SpMP	HDagg
Intel x86	10.79	7.60	3.25
AMD x86	5.20	3.65	1.98
Huawei ARM	9.27	n/a	2.16

Table 7.4. Geometric mean speed-ups relative to Serial of our algorithms over different machines and processor architectures. Experiments were conducted using 22 cores on the SuiteSparse data set.

7.6. Scaling with the number of cores. Another natural question is how our algorithm scales with a growing number of cores. To examine this, we illustrate the speed-ups (over serial execution) for different numbers of cores in Table 7.5. We note that this experiment was conducted on the AMD x86 machine as it has 64 available cores on a single socket. As one sees, additional cores have diminished or negative returns at the higher end of number of cores. A reason for this is the average wavefront size which is a proxy for the amount of parallelism available. If we split the SuiteSparse data set into groups according to their average wavefront size, we see that these groups scale to different number of cores, see Figure 7.2. This shows that our algorithm does scale if the matrices allow for it.

Algorithm	4	16	32	48	56	64
GrowLocal	2.63	4.15	5.34	5.70	5.76	5.85

Table 7.5. Geometric mean of speed-ups relative to Serial of GrowLocal for different number of cores on the AMD x86 machine taken over the SuiteSparse data set.

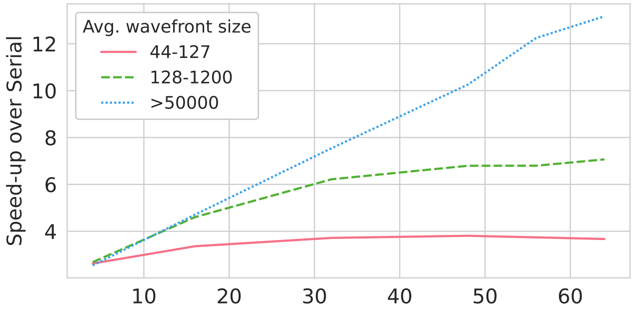


Figure 7.2. Geometric mean speed-ups of GrowLocal for different number of cores on the AMD x86 machine taken over the SuiteSparse data set categorized by average wavefront size.

7.7. Amortization of scheduling time. In this section, we consider the gain of the different scheduling algorithms when the scheduling time is taken into account. We measure the amortization threshold as the following ratio[6]:

$$\frac{\text{scheduling\_time}}{\text{serial\_execution\_time} - \text{parallel\_execution\_time}}\,. \tag{7.1}$$

The same metric was considered by Zarebavani *et. al.* [ZCL+22, §V.B] and expresses how often the schedule needs to be reused in order to justify the time spent on computing it. Table 7.6 presents the amortization threshold for GrowLocal and the two baselines, with the 25th percentile, median, and 75th percentile values shown for each algorithm. The results indicate that the amortization threshold of GrowLocal (both with and without Funnel) is of a similar magnitude to that of SpMP. HDagg has a significantly higher amortization threshold on the SparseSuite data set. However, we remark that, e.g., on the METIS pre-processed data set we observed a comparable median amortization threshold of 44.21 for HDagg.

Algorithm	**Q25**	**Median**	**Q75**
GrowLocal	23.78	26.12	30.28
Funnel+GL	17.78	21.74	27.78
SpMP	3.65	5.51	8.41
HDagg	311.23	961.39	1848.80

Table 7.6. Amortization threshold of several scheduling algorithms on the SuiteSparse data set with the 25th percentile, median, and 75th percentile values shown for each algorithm. The data was collected on the Intel x86 machine using 22 cores.

7.8. Block parallel scheduling. In order to further reduce the amortization threshold of GrowLocal, we now consider the effect of subdividing the matrix into blocks and applying the GrowLocal scheduler on each block in parallel, cf. Section 3.1. In Table 7.7, we record the effect of running GrowLocal with multiple scheduling threads on scheduling time, floating point operations per second, number of supersteps, and amortization threshold. We see that using multiple scheduling threads can lead to super-linear speed-up of scheduling time. This is because there are now several (long) egdes that never have to be considered, cf. Figure 3.1. We also see that the SpTRSV solve time is more significantly affected when using a higher number of scheduling threads. We note that these effects are heavily dependent on the matrix. Some matrices, such as 'af_shell7' and 'bmwcra_1' have approximately a 30% performance drop in the SpTRSV solve time already when using just two blocks, whereas 'bundle_adj' and 'Hook_1498' are hardly affected even when using 16 blocks. Despite the performance drops, we find a near linear decrease in the amortization threshold.

Table 7.7 indicates, for instance, that using 6 scheduling threads is a noteworthy compromise. This lowers the median amortization threshold to 4.54, which is smaller than that of SpMP, cf. Table 7.6, whilst maintaining a 1.05× speed-up over SpMP. This makes GrowLocal superior on both metrics simultaneously.

[6] If the parallel execution is slower than the serial one, then the amortization threshold is defined as $+\infty$.

Threads	**Sched. Time**	**Flops/s**	**Supersteps**	**Amort. Threshold**
1	1.00	1.00	1.00	26.12
2	2.01	0.89	1.47	13.59
4	4.11	0.79	1.99	6.91
6	6.28	0.74	2.35	4.54
8	8.34	0.70	2.66	3.48
16	17.06	0.57	3.84	1.78
22	23.43	0.52	4.53	1.31

Table 7.7. Geometric means of relative speed-up of scheduling time, relative decrease in double precision floating point operations per second, and relative increase of number of supersteps of GrowLocal compared to using just a single scheduling thread, that is, a single scheduling block, together with the median amortization threshold of GrowLocal on the SuiteSparse data set. The data was collected on the Intel x86 machine using 22 cores.

8. Conclusion and future directions

The results show that our GrowLocal scheduler indeed significantly speeds up the parallel SpTRSV kernel, reducing the execution time by a 1.42× geometric-mean factor compared to SpMP and 3.32× compared to HDagg on the SuiteSparse benchmark. The data also shows that the algorithm performs similarly well on multiple other data sets, and the improvements are consistent over various architectures. The scheduling time of GrowLocal is also competitive with the baselines, especially when combined with the block decomposition technique.

Future work may consider the adaptation of our algorithm to non-uniform memory access (NUMA) architectures. In particular, the AMD x86 data in Section 7.6 confirms that our algorithm scales well to a high number of cores. However, when solving SpTRSV on highly NUMA architectures, we expect the parallel execution to be less effective. In order to adapt to such a NUMA setting, one should consider fundamental changes to the SpTRSV kernel and the data structures, such as partitioning the matrix local to threads, interleaving the vector, and reducing the need for global synchronization. It is also an interesting question whether the scheduling algorithm can be efficiently adapted to NUMA, for example, by considering non-uniform bandwidth or latency. To our knowledge, there are currently no scheduling algorithms for SpTRSV that directly account for such NUMA effects.

Another promising direction for future work is to combine our barrier list scheduling algorithms with other approaches that proved successful for SpTRSV in the past. For instance, one could seek to adapt GrowLocal to a semi-asynchronous setting as in SpMP, in order to allow for a more flexible parallel execution. This could allow for further speed-ups on top of our current results.

Appendix A. Tables of matrices

Here we provide some basic statistics of the matrices used in the experiments, cf. Section 6.2.

Matrix	Size	#Non-zeros	Avg. wf
af_0_k101	503,625	9,027,150	74
af_shell7	504,855	9,046,865	135
apache2	715,176	2,766,523	1,077
audikw_1	943,695	39,297,771	203
bmw7st_1	141,347	3,740,507	199
bmwcra_1	148,770	5,396,386	204
bone010	986,703	36,326,514	470
boneS01	127,224	3,421,188	156
boneS10	914,898	28,191,660	386
Bump_2911	2,911,419	65,320,659	283
bundle_adj	513,351	10,360,701	57,039
consph	83,334	3,046,907	139
Dubcova3	146,689	1,891,669	44
ecology2	999,999	2,997,995	500
Emilia_923	923,136	20,964,171	176
Fault_639	638,802	14,626,683	143
Flan_1565	1,564,794	59,485,419	200
G3_circuit	1,585,478	4,623,152	611
Geo_1438	1,437,960	32,297,325	246
hood	220,542	5,494,489	365
Hook_1498	1,498,023	31,207,734	95
inline_1	503,712	18,660,027	287
ldoor	952,203	23,737,339	141
msdoor	415,863	10,328,399	59
offshore	259,789	2,251,231	75
parabolic_fem	525,825	2,100,225	75,117
PFlow_742	742,793	18,940,627	118
Queen_4147	4,147,110	166,823,197	342
s3dkt3m2	90,449	1,921,955	60
Serena	1,391,349	32,961,525	298
shipsec1	140,874	3,977,139	67
StocF-1465	1,465,137	11,235,263	487
thermal2	1,228,045	4,904,179	991

Table A.1. Matrices and statistics from SuiteSparse Matrix Collection [DH11] used for the evaluation. The average wavefront size (Avg. wf) has been rounded down.

Matrix	**Size**	**#Non-zeros**	**Avg. wf**
af_0_k101_metis	503,625	9,027,150	610
af_shell10_metis	1,508,065	27,090,195	1,065
apache2_metis	715,176	2,766,523	47,678
audikw_1_metis	943,695	39,297,771	1,734
bmwcra_1_metis	148,770	5,396,386	473
bone010_metis	986,703	36,326,514	1,326
boneS10_metis	914,898	28,191,660	2,401
bundle_adj_metis	513,351	10,360,701	11,407
cant_metis	62,451	2,034,917	333
consph_metis	83,334	3,046,907	247
crankseg_2_metis	63,838	7,106,348	86
ecology2_metis	999,999	2,997,995	62,499
Emilia_923_metis	923,136	20,964,171	2,107
Fault_639_metis	638,802	14,626,683	1,458
Flan_1565_metis	1,564,794	59,485,419	2,569
G3_circuit_metis	1,585,478	4,623,152	93,263
Geo_1438_metis	1,437,960	32,297,325	2,887
gyro_metis	17,361	519,260	88
hood_metis	220,542	5,494,489	984
Hook_1498_metis	1,498,023	31,207,734	4,059
inline_1_metis	503,712	18,660,027	1,549
ldoor_metis	952,203	23,737,339	4,858
m_t1_metis	97,578	4,925,574	268
msdoor_metis	415,863	10,328,399	1,856
nasasrb_metis	54,870	1,366,097	287
PFlow_742_metis	742,793	18,940,627	1,023
pwtk_metis	217,918	5,926,171	511
raefsky4_metis	19,779	674,195	111
ship_003_metis	121,728	4,103,881	494
shipsec8_metis	114,919	3,384,159	456
StocF-1465_metis	1,465,137	11,235,263	11,446
thermal2_metis	1,228,045	4,904,179	45,483
tmt_sym_metis	726,713	2,903,837	26,915
x104_metis	108,384	5,138,004	306

Table A.2. Matrices and statistics from SuiteSparse Matrix Collection [DH11] symmetrically permuted using the fill-reducing method 'METIS_NodeND' of [KK98]. The average wavefront size (Avg. wf) has been rounded down.

Matrix	**Size**	**#Non-zeros**	**Avg. wf**
af_0_k101_iCh	503,625	9,027,150	195
af_shell7_iCh	504,855	9,046,865	668
apache2_iCh	715,176	2,766,523	79,464
audikw_1_iCh	943,695	39,297,771	138
bmw7st_1_iCh	141,347	3,740,507	340
bmwcra_1_iCh	148,770	5,396,386	89
bone010_iCh	986,703	36,326,514	340
boneS01_iCh	127,224	3,421,188	245
boneS10_iCh	914,898	28,191,660	521
Bump_2911_iCh	2,911,419	65,320,659	1,048
consph_iCh	83,334	3,046,907	78
Dubcova3_iCh	146,689	1,891,669	1,594
ecology2_iCh	999,999	2,997,995	142,857
Emilia_923_iCh	923,136	20,964,171	511
Fault_639_iCh	638,802	14,626,683	422
Flan_1565_iCh	1,564,794	59,485,419	689
G3_circuit_iCh	1,585,478	4,623,152	88,082
Geo_1438_iCh	1,437,960	32,297,325	768
hood_iCh	220,542	5,494,489	1,050
Hook_1498_iCh	1,498,023	31,207,734	649
inline_1_iCh	503,712	18,660,027	679
ldoor_iCh	952,203	23,737,339	3,317
msdoor_iCh	415,863	10,328,399	956
offshore_iCh	259,789	2,251,231	1,114
parabolic_fem_iCh	525,825	2,100,225	19,475
PFlow_742_iCh	742,793	18,940,627	240
Queen_4147_iCh	4,147,110	166,823,197	719
s3dkt3m2_iCh	90,449	1,921,955	104
Serena_iCh	1,391,349	32,961,525	940
shipsec1_iCh	140,874	3,977,139	259
StocF-1465_iCh	1,465,137	11,235,263	2,990
thermal2_iCh	1,228,045	4,904,179	47,232

TABLE A.3. Matrices and statistics from SuiteSparse Matrix Collection [DH11] post Eigen incomplete Cholesky [GJ+10] used for the evaluation. The average wavefront size (Avg. wf) has been rounded down.

Matrix	Size	#Non-zeroes	Avg. wf
ER_100k_19m_A	100,000	19,999,021	109
ER_100k_19m_B	100,000	19,998,182	109
ER_100k_19m_C	100,000	19,997,897	107
ER_100k_19m_D	100,000	19,995,405	106
ER_100k_19m_E	100,000	19,994,516	107
ER_100k_19m_G	100,000	19,989,535	106
ER_100k_19m_H	100,000	19,999,989	110
ER_100k_1m_A	100,000	1,001,528	1,785
ER_100k_1m_B	100,000	1,000,452	1,818
ER_100k_1m_C	100,000	1,000,315	1,818
ER_100k_1m_E	100,000	1,000,044	1,666
ER_100k_1m_F	100,000	1,000,406	1,785
ER_100k_1m_G	100,000	1,001,171	1,724
ER_100k_1m_H	100,000	1,001,551	1,886
ER_100k_1m_I	100,000	1,000,237	1,639
ER_100k_1m_J	100,000	1,001,533	1,851
ER_100k_20m_F	100,000	20,001,732	107
ER_100k_20m_I	100,000	20,006,442	109
ER_100k_20m_J	100,000	20,003,479	109
ER_100k_4m_A	100,000	4,998,205	395
ER_100k_4m_C	100,000	4,999,271	398
ER_100k_4m_G	100,000	4,999,358	401
ER_100k_4m_J	100,000	4,996,501	414
ER_100k_5m_B	100,000	5,006,107	411
ER_100k_5m_D	100,000	5,001,575	404
ER_100k_5m_E	100,000	5,004,251	400
ER_100k_5m_F	100,000	5,002,190	400
ER_100k_5m_H	100,000	5,000,573	409
ER_100k_5m_I	100,000	5,001,846	400
ER_100k_999k_D	100,000	999,915	1,818

Table A.4. Matrices and statistics in the Erdős–Rényi data set used for the evaluation. The average wavefront size (Avg. wf) has been rounded down.

Matrix	Size	#Non-zeroes	Avg. wf
NB_p14_b10_100k_A	100,000	146,565	87
NB_p14_b10_100k_B	100,000	146,328	115
NB_p14_b10_100k_C	100,000	147,201	61
NB_p14_b10_100k_D	100,000	146,972	73
NB_p14_b10_100k_E	100,000	147,369	73
NB_p14_b10_100k_F	100,000	146,855	111
NB_p14_b10_100k_G	100,000	147,350	132
NB_p14_b10_100k_H	100,000	147,412	85
NB_p14_b10_100k_I	100,000	147,132	132
NB_p14_b10_100k_J	100,000	146,781	105
NB_p3_b42_100k_A	100,000	127,045	46
NB_p3_b42_100k_B	100,000	127,019	55
NB_p3_b42_100k_C	100,000	127,708	29
NB_p3_b42_100k_D	100,000	127,341	45
NB_p3_b42_100k_E	100,000	127,569	67
NB_p3_b42_100k_F	100,000	127,137	47
NB_p3_b42_100k_G	100,000	127,774	52
NB_p3_b42_100k_H	100,000	127,029	46
NB_p3_b42_100k_I	100,000	127,475	39
NB_p3_b42_100k_J	100,000	127,275	62
NB_p5_b20_100k_A	100,000	102,053	1,298
NB_p5_b20_100k_B	100,000	102,621	1,063
NB_p5_b20_100k_C	100,000	102,021	1,298
NB_p5_b20_100k_D	100,000	102,968	1,075
NB_p5_b20_100k_E	100,000	102,650	952
NB_p5_b20_100k_F	100,000	102,309	1,162
NB_p5_b20_100k_G	100,000	103,152	892
NB_p5_b20_100k_H	100,000	102,324	1,190
NB_p5_b20_100k_I	100,000	102,465	1,369
NB_p5_b20_100k_J	100,000	102,244	1,010

Table A.5. Matrices and statistics in the narrow bandwidth data set used for the evaluation. The average wavefront size (Avg. wf) has been rounded down.

Appendix B. Time and space complexity of GrowLocal

Below we provide a more detailed discussion of the time and space complexity of GrowLocal. We first restate Theorem 3.1 more formally, including the assumption that the out-degrees and compute weights are within a constant factor.

Theorem 3.1 (formal). Assume that there exist positive constants η and ϱ such that for all vertices $u, v \in V$, we have

$$\omega(u) \leq \eta \cdot \omega(v), \tag{B.1}$$

$$\deg^+(\mathrm{u}) \leq \varrho \cdot |E|/|V|. \tag{B.2}$$

Then, the time complexity of GrowLocal is $O(|E| \cdot \log |V|)$ and the space complexity is $O(|E|)$.

Proof. For each iteration i, denote by $\mathcal{V}_p^{(i)}$ the vertices which have been assigned to core p in iteration i, by $\Gamma_p^{(i)} = |\mathcal{V}_p^{(i)}|$ the number of vertices assigned to core p in iteration i, and by $\Omega_p^{(i)}$ the sum of weights of these vertices. From the algorithm design, specifically Line 8, we have for each iteration i and each core p that

$$\Omega_p^{(i)} \leq \mu \Omega_1^{(i)}, \tag{B.3}$$

for some fixed positive constant μ. We furthermore use the notations

$$\Gamma_{\max}^{(i)} = \max_p \Gamma_p^{(i)}, \tag{B.4}$$

$$\Gamma_{\Sigma}^{(i)} = \sum_p \Gamma_p^{(i)}, \quad \text{and} \tag{B.5}$$

$$\omega_{\min} = \min_{v \in V} \omega(v) \tag{B.6}$$

for simplicity.

The key observation of the analysis is that during the formation of a superstep, the total number of vertices assigned over all the iterations is only a linear factor away from the number of vertices assigned in the final superstep. If we restrict ourselves to the first core, this is easy to see intuitively: the first iteration assigns 20 vertices to core, the next iteration assigns $20 \cdot \frac{3}{2}$, the following assigns $20 \cdot (\frac{3}{2})^2$, and so on. If the iteration that is accepted in the end assigns α^* vertices to the first core, then the preceding iterations assign at most $\alpha^* \sum_{i=1}^{\infty} (\frac{2}{3})^i$ altogether, and the last examined iteration (which is rejected) possibly also assigns $\frac{3}{2}\alpha^*$; this is altogether still in $O(\alpha^*)$. Note that the ratio between the last two iterations can also be less than $\frac{3}{2}$, but this does not affect the claim.

Extending the argument above to all the cores is slightly more technical due to the different vertex weights. For a core p, we have

$$\Gamma_p^{(i)} = \sum_{v \in \mathcal{V}_p^{(i)}} 1 \leq \sum_{v \in \mathcal{V}_p^{(i)}} \eta \frac{\omega(v)}{\sum_{u \in \mathcal{V}_1^{(i)}} \omega(u) / \sum_{u \in \mathcal{V}_1^{(i)}} 1}$$
$$= \eta \frac{\Omega_p^{(i)}}{\Omega_1^{(i)}} \sum_{u \in \mathcal{V}_1^{(i)}} 1 \leq \eta\mu \sum_{u \in \mathcal{V}_1^{(i)}} 1 = \eta\mu\Gamma_1^{(i)}. \tag{B.7}$$

Here, we used (B.1) and (B.3). Thus, if $\alpha^{(i)} = 20 \cdot (\frac{3}{2})^{i-1}$ is the parameter used for iteration i, then we have

$$\alpha^{(i)} \leq \Gamma_{\max}^{(i)} \leq \eta\mu \cdot \alpha^{(i)}. \tag{B.8}$$

This means that for iterations i and j with $i < j$, we get that the ratio $\Gamma_{\max}^{(i)}/\Gamma_{\max}^{(j)}$ is at most $\eta\mu \cdot (\frac{2}{3})^{j-i}$.

For the proof below, we actually require a similar upper bound on the ratio

$$\frac{\Gamma_{\max}^{(i)} + L}{\Gamma_{\max}^{(j)} + L} \tag{B.9}$$

instead. For this, we separate two cases, namely the first few iterations and all the remaining ones. Recall that L was chosen as a constant. Assume first that we have $\Gamma_{\max}^{(i)} \geqslant L$. In this case, we can upper bound the expression above by

$$\frac{2 \cdot \Gamma_{\max}^{(i)}}{\Gamma_{\max}^{(j)}} \leqslant 2\eta\mu \cdot \left(\tfrac{2}{3}\right)^{j-i} . \tag{B.10}$$

On the other hand, assume that $\Gamma_{\max}^{(i)} < L$. In this case, we can upper bound (B.9) simply by 1. The key observation is that the number of these iterations with $\Gamma_{\max}^{(i)} < L$ is at most a constant in any superstep. Indeed, starting with $\alpha = 20$ and multiplying by $\frac{3}{2}$ each round, we already have $\alpha > L$ by the $\lceil \log(L)/\log(1.5) \rceil$-th iteration, and hence $\Gamma_{\max}^{(i)} \geqslant L$ for $i \geq C_L \coloneqq \lceil \log(L)/\log(1.5) \rceil$. As such, there are only $C_L = O(\log(L)) = O(1)$ distinct values where we will use this upper bound of 1.

Our algorithm only accepts and saves an iteration if its parallelization rate β is at least a 0.97 factor of the best parallelization rate observed so far during this superstep. Hence, if iteration j is accepted and i is any iteration such that $i < j$, then we have

$$\frac{\sum_p \Omega_p^{(j)}}{\max_p \Omega_p^{(j)} + L} \geqslant 0.97 \cdot \frac{\sum_p \Omega_p^{(i)}}{\max_p \Omega_p^{(i)} + L} . \tag{B.11}$$

Due to our assumption on the vertex weights, we have for any iteration i that

$$\omega_{\min} \cdot \Gamma_{\max}^{(i)} \leqslant \max_p \Omega_p^{(j)} \leqslant \eta\omega_{\min} \cdot \Gamma_{\max}^{(i)} \tag{B.12}$$

as well as

$$\omega_{\min} \cdot \Gamma_{\Sigma}^{(i)} \leqslant \sum_p \Omega_p^{(i)} \leqslant \eta\omega_{\min} \cdot \Gamma_{\Sigma}^{(i)}. \tag{B.13}$$

Using these in Inequality (B.11), we get that

$$\frac{\eta\omega_{\min} \cdot \Gamma_{\Sigma}^{(j)}}{\omega_{\min} \cdot \Gamma_{\max}^{(j)} + L} \geqslant 0.97 \cdot \frac{\omega_{\min} \cdot \Gamma_{\Sigma}^{(i)}}{\eta\omega_{\min} \cdot \Gamma_{\max}^{(i)} + L} , \tag{B.14}$$

which further implies

$$\Gamma_{\Sigma}^{(i)} \leqslant \frac{\eta^2}{0.97} \cdot \frac{\Gamma_{\max}^{(i)} + L}{\Gamma_{\max}^{(j)} + L} \cdot \Gamma_{\Sigma}^{(j)} . \tag{B.15}$$

Using our upper bounds on (B.9) and $\eta = O(1)$, this implies

$$\Gamma_{\Sigma}^{(i)} \leqslant O(1) \cdot \Gamma_{\Sigma}^{(j)} \tag{B.16}$$

for $i \in \{1, ..., C_L - 1\}$, and

$$\Gamma_{\Sigma}^{(i)} \leqslant O(1) \cdot \left(\tfrac{2}{3}\right)^{j-i} \cdot \Gamma_{\Sigma}^{(j)} \tag{B.17}$$

for $i \geqslant C_L$. Assuming that iteration j is the final worthy iteration that is accepted for our superstep, this means that the total number of assigned vertices made in iterations $i \in \{1, 2, ..., j-1\}$ is at most

$$\sum_{i=1}^{j-1} \Gamma_{\Sigma}^{(i)} \leqslant O(1) \cdot \Gamma_{\Sigma}^{(j)} \cdot \left(O(1) + \sum_{\ell=1}^{j-1} \left(\tfrac{2}{3}\right)^{\ell} \right) . \tag{B.18}$$

With the geometric sum upper bounded by 2, we get that the number of assigned vertices is indeed in $O(\Gamma_{\Sigma}^{(j)})$.

Note that the argument above does not consider the possible last iteration $(j+1)$ which is rejected by our algorithm. Nevertheless, we can bound the assignments here with a similar argument. If the iteration was rejected, then its parallelization score is at most as high as that of iteration j, i.e., $\beta^{(j+1)} \leqslant \beta^{(j)}$. As before, this implies

$$\frac{\omega_{\min} \cdot \Gamma_{\Sigma}^{(j+1)}}{\omega_{\min} \cdot \eta \cdot \Gamma_{\max}^{(j+1)} + L} \leqslant \frac{\omega_{\min} \cdot \eta \cdot \Gamma_{\Sigma}^{(j)}}{\omega_{\min} \cdot \Gamma_{\max}^{(j)} + L}, \tag{B.19}$$

and hence

$$\Gamma_{\Sigma}^{(j+1)} \leqslant O(1) \cdot \frac{\Gamma_{\max}^{(j+1)} + L}{\Gamma_{\max}^{(j)} + L} \cdot \Gamma_{\Sigma}^{(j)}. \tag{B.20}$$

As

$$\begin{aligned} \frac{\Gamma_{\max}^{(j+1)} + L}{\Gamma_{\max}^{(j)} + L} &= \frac{\Gamma_{\max}^{(j+1)} - \Gamma_{\max}^{(j)}}{\Gamma_{\max}^{(j)} + L} + 1 \\ &\leqslant \frac{\Gamma_{\max}^{(j+1)}}{\Gamma_{\max}^{(j)}} + 1 \leqslant \tfrac{3}{2}\eta\mu + 1, \end{aligned} \tag{B.21}$$

we have that $\Gamma_{\Sigma}^{(j+1)}$ is again in $O(\Gamma_{\Sigma}^{(j)})$.

As such, the number of assignments in any superstep is linear in the size of the vertices that are finally scheduled. Summing this up over all the supersteps, we get that the algorithm altogether only makes $O(|V|)$ assignments over all supersteps and iterations.

For each assignment, the chosen vertex is selected from a priority queue data structure. Each such data structure contains at most $|V|$ vertices, so the cost of each assignment is $O(\log|V|)$. However, after each assignment of a concrete vertex v, we also need to examine all the children u of v, check if u also becomes ready with this assignment (i.e., all parents of u are computed now), and if so, then also insert u into such a priority queue at a time cost of $O(\log|V|)$. Since any vertex v has at most $\varrho \cdot |E|/|V|$ children and $\varrho \in O(1)$, this sums up to a total of $O(|V| \cdot |E|/|V| \cdot \log|V|)$ over all the $O(|V|)$ assignments. This results in an overall time complexity of $O(|E| \cdot \log|V|)$ for the algorithm.

The space complexity of the algorithm is much easier to settle: each iteration only stores $O(|V|)$ data, and we store at most two iterations at a time, so the main bottleneck here is simply storing the input DAG itself, which requires $O(|E|)$ space. □

For the sake of completeness, we complement the theoretical bound in Theorem 3.1 with empirical data in Figure B.1.

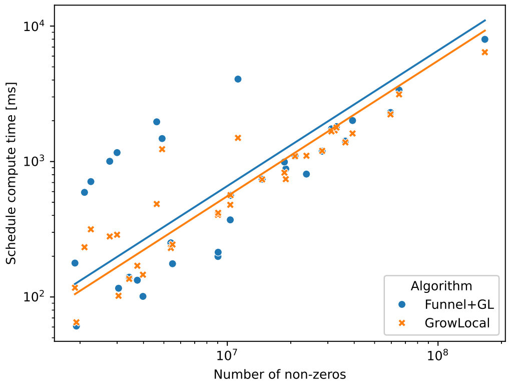


FIGURE B.1. Scheduling time of Funnel+GL and GrowLocal on the SuiteSparse data set. The straight lines are the best square-mean-error fit of the family of curves $\log(y) = \log(x) + c$, where c is a free parameter.

Appendix C. Further discussion

C.1. Comparison to barrier list schedulers. Recall that one of the recent works on DAG scheduling with synchronization barriers is that of Papp *et al.* [PAKY24], which analyzes schedulers not for a concrete application, more abstractly in terms of BSP cost. While the so-called BSPg scheduling heuristic in this work is rather different from our algorithm, the idea of prioritizing vertices in GrowLocal that are computable exclusively on a specific core was inspired by this algorithm. To show for completeness that our scheduler also significantly outperforms this BSPg algorithm, we also ran this BSPg scheduling algorithm as a baseline. The results show that GrowLocal achieves a factor 8.31× geometric-mean speed-up to BSPg on the SutieSparse data set.

C.2. On the synchronisation parameter L. Recall that the parameter L in GrowLocal, Algorithm 3.1, represents the time cost of inserting a synchronization barrier, and is used to determine the parallelization rate in our GrowLocal algorithm.

If we consider the compute time of basic operations (additions or multiplications) with double precision numbers, and compare this to the time of synchronization, we get that the correct magnitude of L ranges from a few hundreds to a few thousands on modern computing architectures. We ran some preliminary experiments with a few different choices of L on this order of magnitude, and chose a value of $L = 500$ based on these empirical observations.

References

[ACD74] Thomas L. Adam, K. Mani Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[AS89] Edward Anderson and Youcef Saad. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing*, 1(01):73–95, 1989.

[AS93] Fernando L. Alvarado and Robert Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM Journal on Scientific Computing*, 14(2):446–460, 1993.

[AYU21] Najeeb Ahmad, Buse Yilmaz, and Didem Unat. A split execution model for sptrsv. *IEEE Transactions on Parallel and Distributed Systems*, 32(11):2809–2822, 2021.

[BLM$^+$24] Toni Böhnlein, Benjamin Lozes, Christos Matzoros, Pál András Papp, and Raphael S. Steiner. OneStopParallel. https://github.com/Algebraic-Programming/OneStopParallel, 2024.

[Che22] Kazem Cheshmi. *Transforming Sparse Matrix Computations*. PhD thesis, University of Toronto, Computer Science, 2022.

[CKSD17] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. Sympiler: Transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 13:1–13:13, New York, NY, USA, 2017. ACM.

[CKSD18] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. ParSy: inspection and transformation of sparse matrix computations for parallelism. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 779–793. IEEE, 2018.

[CLB94] Jason Cong, Zheng Li, and Rajive Bagrodia. Acyclic multi-way partitioning of boolean networks. In *Proceedings of the 31st annual design automation conference*, pages 670–675, 1994.

[DH11] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[DM02] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91:201–213, 2002.

[ER59] Paul Erdős and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae*, 6:290–297, 1959.

[FER$^+$13] Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, Jagannathan Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):1–29, 2013.

[GJ$^+$10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[Gra69] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[HCAL89] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *siam journal on computing*, 18(2):244–257, 1989.

[HKSL14] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 166–177, 2014.

[HKU$^+$17] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Ümit V. Çatalyürek. Acyclic partitioning of large directed acyclic graphs. In *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, pages 371–380. IEEE, 2017.

[IKS75] Tadakatsu Ishiga, Tokinori Kozawa, and Shoji Sato. A logic partitioning procedure by interchanging clusters. In *Proceedings of the 12th Design Automation Conference*, pages 369–377, 1975.

[Kah62] Arthur B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[KAKS97] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. In *Proceedings of the 34th annual Design Automation Conference*, pages 526–529, 1997.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[LLH$^+$16] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S. Duff, and Brian Vinter. 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*, pages 617–630. Springer, 2016.

[LNL20] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. Efficient block algorithms for parallel sparse triangular solve. In *Proceedings of the 49th International Conference on Parallel Processing*, pages 1–11, 2020.

[May09] Jan Mayer. Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing*, 86:291–312, 2009.

[MSQ03] Shang Mingsheng, Sun Shixin, and Wang Qingxian. An efficient parallel scheduling algorithm of dependent task graphs. In *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 595–598. IEEE, 2003.

[PA92] Alex Pothen and Fernando L. Alvarado. A fast reordering algorithm for parallel sparse triangular solution. *SIAM journal on scientific and statistical computing*, 13(2):645–653, 1992.

[PAKY24] Pál András Papp, Georg Anegg, Aikaterini Karanasiou, and Albert-Jan N. Yzelman. Efficient Multi-Processor Scheduling in Increasingly Realistic Models. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2024.

[PSSD14] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 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*, pages 124–140. Springer, 2014.

[PSSS21] Merten Popp, Sebastian Schlag, Christian Schulz, and Daniel Seemaier. Multilevel Acyclic Hypergraph Partitioning. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–15. SIAM, 2021.

[RG92] Edward Rothberg and Anoop Gupta. Parallel ICCG on a hierarchical memory multiprocessor—addressing the triangular solve bottleneck. *Parallel Computing*, 18(7):719–741, 1992.

[RVG02] Andrei Radulescu and Arjan J. C. Van Gemund. Low-cost task scheduling for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):648–658, 2002.

[Sal90] Joel H. Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM journal on scientific and statistical computing*, 11(1):123–144, 1990.

[Sch20] Sebastian Schlag. *High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2020. 46.12.02; LK 01.

[SMB88] Joel H. Saltz, Ravi Mirchandaney, and Doug Baxter. Run-time parallelization and scheduling of loops. Technical report, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1988.

[TW67] William F. Tinney and John W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, 1967.

[Val90a] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[Val90b] Leslie G. Valiant. General purpose parallel architectures. In *Algorithms and Complexity*, pages 943–971. Elsevier, 1990.

[WS18] Huijun Wang and Oliver Sinnen. List-scheduling versus cluster-scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 29(8):1736–1749, 2018.

[YSAU20] Buse Yılmaz, Buğrra Sipahioğrlu, Najeeb Ahmad, and Didem Unat. 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*, pages 188–198, 2020.

[ZCL$^+$22] Behrooz Zarebavani, Kazem Cheshmi, Bangtian Liu, Michelle Mills Strout, and Maryam Mehri Dehnavi. HDagg: hybrid aggregation of loop-carried dependence iterations in sparse matrix computations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1217–1227. IEEE, 2022.