EFFICIENT PARALLEL SCHEDULING FOR SPARSE TRIANGULAR SOLVERS

TONI BÖHNLEIN, PÁL ANDRÁS PAPP, RAPHAEL S. STEINER, AND ALBERT-JAN N. YZELMAN[†] {toni.boehnlein; pal.andras.papp; raphael.steiner; albertjan.yzelman}@huawei.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, which we call barrier list scheduling, 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 $3.24 \times$ and $1.45 \times$ geometric-mean speed-up, respectively. We achieve this by obtaining an up to $11 \times$ 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.

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 subjects. 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.

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 mark 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.

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

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

Date: March 10, 2025.

Key words and phrases. Sparse triangular linear system solve, SpTRSV, SpTRSM, forward- and backwardsubstitution algorithm, barrier list scheduler, synchronous parallel algorithm.

[†]Authors are listed in alphabetical order.



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.

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 1.1(B), 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 put forth barrier list schedulers as an effective synchronous scheduling algorithm archetype for solving sparse triangular linear systems. In particular, we present and analyze two specific algorithms that are both tailored towards the SpTRSV application. In our experiments, we establish that these algorithms produce significantly superior parallel schedules compared to the baseline methods. Specifically, our *Funnel Locking* algorithm achieves a reduction in execution time of $1.45 \times$ compared to SpMP and of $3.24 \times$ compared to HDagg, on the SuiteSparse Matrix Collection benchmark [DH11] used by previous studies, see Figure 1.2. On uniformly random matrices, our other algorithm, called *Funnel p-ivotal path*, achieves a similar speed-up of $1.62 \times$ compared to SpMP and $1.87 \times$ compared to HDagg in execution time.

Our algorithms obtain these speed-ups by significantly reducing the number of synchronization barriers required: we report an up to $11 \times$ reduction in the number of barriers relative to



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

HDagg, whilst maintaining a good workload balance. The results also show that our schedulers provide consistent improvements over several different computing architectures and types of input matrices. The scheduling algorithms themselves also have a just above linear time complexity, making them viable tools for various applications.

In summary, the main contributions of our paper are:

- two novel algorithms for generating efficient parallel schedules for SpTRSV execution,
- adaptations and extensions of previous coarsening and reordering techniques that enhance the schedules,
- a short theoretical proof showing that our DAG coarsening techniques preserve acyclicity, based on the new concept of cascades, and
- experiments confirming that the above schedulers achieve significant speed-ups over the SpMP and HDagg baselines, on various architectures and data sets, including orthogonal testing of the individual techniques proposed.

1.1.1. Barrier list schedulers. There is an extensive amount of prior work on parallel DAG scheduling in the literature. A recent survey classifies the most relevant algorithms into two major groups, list schedulers and cluster schedulers [WS18]. List schedulers [Gra69, ACD74, HCAL89, RVG02, MSQ03] typically assign priorities to each vertex and schedule them in a topological order according to these. Such algorithms are often more efficient when the number of cores is limited [WS18].

The idea of list schedulers has also been adapted recently to a setting with barrier synchronization [PAKY24]. We refer to this new approach as *barrier list scheduling*. Similarly to list scheduling, these algorithms maintain a set of vertices that are ready to be executed, i.e., all their parents have been computed. Whenever a core p becomes free, they assign a ready vertex to p that does not require new communication between cores, with a preference for vertices that can only be executed on the core p. The schedulers only introduce a new synchronization barrier when a given fraction of cores become idle.

In our paper, we present two barrier-list-scheduling algorithms for finding efficient parallel schedules for the SpTRSV kernel. We extend the prototype of the barrier-list-scheduling idea by Papp *et al.* [PAKY24] with various improvements: we refine the rule for inserting a synchronization barrier, we decrease the amount of compute resources wasted before the synchronization barriers, and we apply more advanced data structures for an efficient implementation. More importantly, the key to the behavior of a barrier-list-scheduling algorithm is the priority function, which selects the next vertex to be assigned to a free core whenever there are multiple options. In our two schedulers, these priority functions are tailored specifically towards the SpTRSV application:

- the *p*-ivotal path scheduler aims to prioritize vertices with long outgoing paths and/or a high-number of successors in the DAG, whilst
- the Locking scheduler aims to ensure that we can compute the highest possible number of vertices before having to insert a new synchronization barrier.

While both algorithms are similar, our experiments show that Locking performs better on the SuiteSparse Matrix Collection and p-ivotal path is superior on the (unstructured) random data sets.

1.1.2. Coarsening. We combine our scheduling algorithms with a DAG coarsening approach to obtain significantly smaller DAGs, whilst preferably retaining most of the original structure. Our schedulers can then be applied to the coarse graph. The resulting schedule is subsequently pulled back to the original graph to obtain the final schedule. Such a coarsening technique is widely applied in graph partitioning and scheduling tools [KAKS97, PSSS21] and has various advantages for our scheduler: it greatly reduces the size of the graph, improves data locality, and can also help reduce the number of synchronization steps.

In order to produce a valid scheduling problem, the coarsening needs to ensure that it preserves the acyclicity of the DAG. Methods that fulfill this property have been studied in several works before, see, for example, [CLB94, FER⁺13, HKU⁺17, ZCL⁺22] and references therein. In Section 4, we propose 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.

1.1.3. *Reordering.* Besides the scheduling 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.

1.2. Additional related work. An optimization method for parallel SpTRSV execution, orthogonal to what we have mentioned so far, is to break the lower triangular matrix into blocks and apply different algorithms and allocation of cores to each block individually depending on block type and sparsity pattern [AS89, May09, AYU21, YSAU20]. The 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. This has been particularly impactful for GPU implementations [LLH⁺16, LNL20]. Our scheduling algorithms would naturally fit in as a building block for such optimizations.

Finally, 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].

1.3. Acknowledgements. We would like to thank Weifeng Liu and Yves Baumann for stimulating conversations on this and surrounding topics.

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 $\{u \mid (u, v) \in E\}$ and $\{u \mid (v, u) \in E\}$ are called the parents of v and the children of v, respectively. The in- and out-degree of v are the number of parents and children of v, respectively. 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 $A = (A_{i,j})_{i,j=1,...,n} \in \mathbb{R}^{n \times n}$, 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

$$x_{i} = \frac{1}{A_{i,i}} \left(b_{i} - \sum_{j=1}^{i-1} A_{i,j} x_{j} \right).$$
(2.1)

The 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 SpTSRSV and 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 $(u, v) \in E$:

- $\sigma(u) \leq \sigma(v);$
- 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 and dictate 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. Barrier list schedulers

The idea of adapting list schedulers to a setting with communication barriers has been explored recently by Papp *et al.* [PAKY24] in the context of abstract BSP scheduling. The barrier-list-scheduler prototype in their paper behaves similarly to a classical list scheduler, assigning concrete time steps to the starting and finishing times of each task. Whenever a core p becomes free, it prioritizes assigning a next vertex to this core from those that are only computable on p in the current superstep, since some of its parents were computed on p in the current superstep. When half of the cores become idle (i.e., cannot be assigned new vertices without communication), a synchronization barrier is inserted.

The skeleton of our schedulers is similar to the base idea of this algorithm, but with many of the ingredients improved. In particular, the essence of the algorithm lies in the *priority function* that selects the next vertex to assign to a free core when there are multiple vertices ready. We present two schedulers with priority functions tailored towards the problem of developing work-balanced schedules with few synchronization steps. Both of our schedulers define this priority function via assigning priority scores to every vertex that is computable in the current superstep. This score may be core-specific as a vertex may be computable on more than one core. Whenever a core becomes free, our algorithms select the vertex with the highest score for the corresponding core as the next vertex to compute on this core.

• In the *p*-ivotal path scheduler, we attempt to prioritize vertices that have long outgoing paths and/or a high number of successors in the (vertex-weighted) DAG $G = (V, E, \omega)$. For this, we use a static scoring function. The score of each vertex $v \in V$ is computed recursively using the parameter $p \geq 1$:

$$\operatorname{prio}[v] = \omega(v) + \left(\sum_{u \in \operatorname{Children}(v)} \operatorname{prio}[u]^p\right)^{\frac{1}{p}}.$$
(3.1)

Note that choosing $p \to \infty$ is just the longest path to a sink vertex excluding communication, which is sometimes referred to as bottom (vertex) distance. In our implementation, we have simply chosen p = 2. We remark that one needs to be wary of numerical instability (overflow) in the implementation of this priority.

In the Locking scheduler, we aim to specifically avoid vertices that would lock out some other vertex from the current superstep due to a communication requirement. In particular, the assignment of vertex v to a core p gets a penalty value of l if there are l yet-uncomputed children of v that already have parents computed on (at most one) core other than p in the current superstep, i.e., assigning v to p would imply that these l vertices can be computed in the next superstep at the earliest. This strategy aims to increase the number of vertices we can compute before having to insert a synchronization barrier. Besides this, to the score of each vertex, we also add the length of the longest outgoing path normalized to a small range [0, 20] as a default initialization value, in order to prioritize progress on the critical paths in general. Note that in this scheduler, whenever a vertex is assigned to a core, the priority values on nearby vertices need to be updated to correctly reflect the new number of vertices that would be locked out by each potential assignment. It is thus of dynamic nature.

Our algorithms use dynamically updated priority queues to ensure that the scoring only imposes a minimal overhead on the scheduling.

Besides this, our schedulers are also extended with a *fill-up* strategy: whenever the algorithm decides that a synchronization barrier is required at time t, we specifically check for each core p whether some lower-weight vertices could still fit into the current computation phase on p without delaying the synchronization barrier, and compute as many of these vertices as we can in order of their priority. This is especially useful for DAGs obtained after coarsening, where the compute weights of vertices can differ significantly.

Finally, our schedulers also use a much more sophisticated rule to decide when to close the current superstep. The fraction α of cores that need to be idle to insert a barrier is changed to a parameter α , chosen between 0.2 and 0.4 for our algorithms. Moreover, our schedulers ensure that even when this threshold is met, our algorithm only inserts a barrier when the current part of the DAG indeed allows for enough parallelization in order to provide work for idle cores.

Algorithm 3.1: Skeleton of our scheduling algorithms

Data: A vertex-weighted DAG $G = (V, E, \omega)$, a set of cores P, and a parameter $\alpha \in [0, 1].$ **Result:** A schedule consisting of processor assignment $\pi: V \to P$ and superstep assignment $\sigma: V \to \mathbb{Z}_{>0}$. 1 superstep $\leftarrow 1$ 2 endStep \leftarrow false 3 finishTimes $\leftarrow \{0\}$ 4 free[p] \leftarrow true, for all $p \in P$ 5 prio[v], prio[p, v] \leftarrow initialize priority, for all $v \in V$ and $p \in P$ 6 ready_{pool}, ready_{all} \leftarrow sources of G 7 ready_p $\leftarrow \emptyset$, for all $p \in P$ 8 while \exists unassigned vertex do if endStep and finishTimes = \emptyset then 9 ready_p $\leftarrow \emptyset$, for all $p \in P$ 10 $\mathrm{ready}_{\mathrm{all}} \gets \mathrm{ready}_{\mathrm{pool}}$ 11 $superstep \leftarrow superstep + 1$ 12 $endStep \leftarrow false$ 13finishTimes $\leftarrow \{0\}$ 14 $t \leftarrow earliest time from finishTimes$ 15for all vertices v that finish at t do 1617free[$\pi(v)$] \leftarrow true for all children u of v do 18if all parents of u are finished then 19 $ready_{pool}.insert(u)$ 20if $\forall (w, u) \in E : (\pi(w) = \pi(v) \text{ or } \sigma(w) < \text{superstep})$ then 2122ready_{$\pi(v)$}.insert(u) 23if ¬endStep or fill-up possible then while $\exists p \text{ with free}[p] = \mathbf{true and } \operatorname{ready}_p \cup \operatorname{ready}_{all} \neq \emptyset \mathbf{do}$ 2425if ¬endStep then $p, v \leftarrow choose \ using \ priority$ 2627else $p, v \leftarrow$ choose using fill-up strategy 28 **delete** v from ready_{pool}, ready_p, and ready_{all} 2930 $\pi(v) \leftarrow p$ $\sigma(v) \leftarrow \text{superstep}$ 31finishTimes.insert(t + w(v))32 $free[p] \leftarrow false$ 33update priority in v's neighborhood // only in Locking 34if α fraction of cores idle and ready_{pool} large enough then 35 $endStep \leftarrow true$ 36finishTimes.remove(t)37 38 return (π, σ)

Altogether, the run time of both of our schedulers is just above linear in the size of the input graph, i.e., in the number of non-zero entries in the matrix. In particular, the worst-case theoretical time complexity is $O(|V| \log |V| + |E|)$ for *p*-ivotal path, and $O(k|V| \log |V| \cdot \sum_{v \in V} \deg(v)^2 + |E|)$ for Locking, where *k* is the number of cores. In practice, our experiments show that the run time of the two schedulers is similar in most cases, and also comparable to that of the HDagg baseline, cf. Section 7.7. In terms of space complexity, *p*-ivotal path requires O(|V| + |E|), whilst Locking requires O(k|V| + |E|) memory.

We provide a detailed pseudocode for our schedulers in Algorithm 3.1, and note:

- The ready_{all} and ready_p structures defined in Lines 6, 7 of Algorithm 3.1 are priority queues that are always sorted according to the priority values prio[v] and prio[p, v] of the contained vertices v, respectively. It is due to the insertions/deletions from these priority queues that the run time of the schedulers is not strictly linear.
- In Line 35 of Algorithm 3.1, the last condition is specifically that

$$|\text{ready}_{\text{pool}}| \ge \min\left(1.2 \cdot \text{busy, busy} + \frac{1}{2} \cdot \text{idle}\right),$$
(3.2)

with 'idle' and 'busy' denoting the current number of idle and busy cores, respectively. The condition guarantees that inserting a synchronization barrier indeed allows us to employ more cores. Intuitively, the first and second term in the minimum ensure that the increase in parallelism is significant in the cases when 'busy' is a small and large value, respectively.

- Once we decide to insert a barrier (the endStep variable is set), the finishing time t_{end} of the current superstep is finalized as the latest finishing time of vertices that are currently being computed, i.e., the latest entry in finishTimes. Then, until the superstep ends, the fill-up strategy is used: we only start computing a vertex v on a core p if this extra computation is still finished by t_{end} at the latest. In Line 28, for each core p, we again select v as the highest-priority vertex in ready_p or ready_{all} that satisfies this condition.
- In Line 5, the *p*-ivotal path scheduler initializes all vertices of the DAG in linear time using Equation (3.1), via a dynamic programming approach that follows a reverse topological ordering.
- In contrast to this, in Line 5, the Locking scheduler also uses dynamic programming to compute the longest outgoing path from each vertex, then normalizes this to the range [0, 20], and uses it as a default priority value. Then, whenever a new vertex is added to the priority queues (in Lines 6, 11, or 22), the rest of the priority function (the number of vertices locked out from the superstep by the assignment) is computed and added to this base priority value.
- In the Locking scheduler, Line 34 considers all children u of v that are not yet locked out of the current superstep. If no parent of u is in the current superstep yet, then for all cores except p, the (ready) parents of u get an extra unit of penalty for locking u out from the superstep. If u already has parents on (only one) other core than p in the current superstep, then it is our current assignment that locks u out of the superstep, so all the (ready) parents of u lose a unit of penalty.

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 we 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 vertexweighted 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 13 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 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.

Finally, we remark that we have observed experimentally that coarsening along in-funnels works better than out-funnels. Whether this has something to do with the graphs in the data set or there is an underlying conceptual reason for this is left as an open question. Algorithm 4.1: In-funnel graph coarsening.

// Main algorithm **Data:** A directed acyclic graph G = (V, E). **Result:** A partition P such that every set $U \in P$ is an in-funnel. 1 Algorithm Funnel(G): $\mathbf{2}$ Partition $\leftarrow \emptyset$ 3 Visited[v] \leftarrow false, $\forall v \in V$ for $v \in V$ in reverse topological order do 4 if Visited[v] then continue 5 $U \leftarrow \{v\}$ 6 $(U,) \leftarrow \text{FunnelDFS}(G, U, v, \emptyset)$ 7 // see subroutine, Line 12 for $u \in U$ do 8 $\bigcup \text{Visited}[u] \leftarrow \mathbf{true}$ 9 Partition.insert(U)1011return Partition // Depth-first search invoked by the main algorithm **Data:** A directed acyclic graph G = (V, E), a non-empty subset $U \subseteq V$ of vertices, a vertex $u \in U$, and a hash map IncludedChildren. **Result:** A non-empty subset $W \subseteq V$ containing U and a hash map IncludedChildren. 12 Subroutine FunnelDFS(G, U, u,IncludedChildren): 13for $v \in Parents(u)$ do if IncludedChildren.find(v) then 14IncludedChildren[v] \leftarrow IncludedChildren[v] + 11516 else IncludedChildren[v] $\leftarrow 1$ 17if IncludedChildren[v] = OutDegree(v) then 18 19 U.insert(v) $(U, \text{IncludedChildren}) \leftarrow \text{FunnelDFS}(G, U, v, \text{IncludedChildren})$ 20return (U, IncludedChildren) 21

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 preferably 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 subDAG). 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 algorithms. Our implementations are available in the *OneStopParallel* repository [BLPS24] 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=close and OMP_PLACES=cores.

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 an untimed execution precedes 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, matrix in the 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 using the optimization flag -03.

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]. This data set is 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 designed to be harder to parallelize.

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 also indicated for each matrix in the overview of the data sets in the appendices.

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 operations¹ 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.

¹The number of floating point operations is equal to twice the number of non-zeros minus the dimension of the matrix.

Remark 6.1. Zarebavani et al. $[ZCL^+22]$ also use the real symmetric positive definite matrices from the SuiteSparse Matrix Collection for their data set. Their data set, however, differs from ours insofar as they first symmetrically permute the sparse symmetric matrix using a fillreducing method of METIS [KK98] and only then take the lower triangular part. In general, this results in non-equivalent SpTRSV problems. As a by-product of this extra step, the matrices in their data set have significantly higher (14× geometric-mean) average wavefront size than the lower triangular part of the original matrices, compare Table A.4 with Table A.1. As a result, the performance characteristics of HDagg and SpMP reported in their paper differ significantly from our experiments.

We mention that in some contexts, such as with the incomplete Cholesky preconditioned conjugate gradient method for sparse symmetric solve, one *is* allowed to first symmetrically permute to increase the average wavefront size.

6.2.2. 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 \pm independently uniformly random². 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.2.

6.2.3. Narrow bandwidth. Unlike the Erdős–Rényi random matrices, we let the lower triangular matrix entry (i, j), with i > j, being 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 6.2.2.

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.3.

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;
- 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;
- 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.

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 algorithms 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 Funnel Locking algorithm achieved a geometric-mean speed-up of $1.45 \times$ compared to SpMP and $3.24 \times$ compared to HDagg. The data shows a similarly large improvement on the Erdős–Rényi data set. Here, the *p*-ivotal path algorithm achieved the best results, with a speed-up of $1.62 \times$ over SpMP and $1.87 \times$ over HDagg.

 $^{^{2}}$ The change of distribution on the diagonal is to avoid numerical instability, in particular divisions by zero.

The narrow bandwidth graphs paint a slightly different picture. Here, the best schedule was produced by SpMP, with our algorithms falling behind. This is because these matrices allow for significantly less parallelizaton by design; such a setting is not a good fit for our schedulers, which aim to utilize most of the cores. In particular, Table 7.2 shows that when only fewer cores are available, our algorithms again outperform the baselines on this data set.

Data set	(Funnel) Locking	(Funnel) <i>p</i> -ivotal path	\mathbf{SpMP}	HDagg
SuiteSparse	10.70 / 9.35	10.44 / 9.27	7.39	3.30
Erdős–Rényi	14.08 / 14.09	14.92 / 15.00	9.23	7.96
Narrow bandw.	2.79 / 2.40	3.25 / 2.53	4.37	1.09

TABLE 7.1. Geometric mean of speed-ups over serial execution of p-ivotal path and Locking with/without Funnel coarsening, compared to the baselines SpMP and HDagg on the Intel x86 machine using 22 cores taken over the data sets described in Section 6.2.

Data set	(Funnel) Locking	(Funnel) <i>p</i> -ivotal path	\mathbf{SpMP}	HDagg
Narrow bandw.	4.16 / 3.87	4.60 / 4.36	3.67	1.96

TABLE 7.2. Geometric mean of speed-ups over serial execution of *p*-ivotal path and Locking with/without Funnel coarsening, compared to the baselines SpMP and HDagg on the Intel x86 machine using 8 cores taken over the narrow bandwidth data set.

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 algorithms are not only faster in execution time on average but are so throughout the diverse SuiteSparse data set.



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

7.2. Fewer synchronization barriers. The result in Table 7.1 show that our schedulers 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 7.3 shows the number of synchronization barriers relative to the number of wavefronts in our algorithms and HDagg. The data indicates a large, up to $14\times$, reduction of number of synchronization barriers compared to the number of wavefronts. This is a reduction of up to $11\times$ compared to HDagg, which explains the significant speed-ups achieved by our methods.

Data set	(Funnel) Locking	(Funnel) <i>p</i> -ivotal path	HDagg
SuiteSparse	14.00 / 9.63	13.26 / 9.76	1.24
Erdős–Rényi	2.75 / 2.74	3.29 / 3.26	1.25
Narrow bandw.	4.53 / 4.01	6.08 / 4.01	1.10

TABLE 7.3. Geometric mean of the reduction of the number of synchronization barriers relative to the number of wavefronts of the matrix.

7.3. Comparison of our algorithms. To compare our techniques and algorithms, we compute the geometric-mean speed-ups over Serial. We present average and interquartile ranges in Figure 7.2. The data shows that the Locking algorithms slightly outperform the *p*-ivotal path ones on the SuiteSparse data set, while the *p*-ivotal path algorithms have an edge on the Erdős–Rényi and the narrow bandwidth data sets. We observe that applying our coarsening algorithm, Funnel, greatly improves results on the SuiteSparse and the narrow bandwidth data sets. However, on the less structured matrices in the Erdős–Rényi data set, coarsening shows no performance gain.



FIGURE 7.2. Geometric mean and interquartile ranges of speed-ups of (Funnel) *p*-ivotal path and (Funnel) Locking relative to Serial grouped according to data set. Experiments were conducted on the Intel x86 machine using 22 cores.

7.4. Impact of reordering. We separately analyze the impact of the reordering step on the performance. Table 7.4 compares the speed-ups achieved by our algorithms with and without the reordering component from Section 5. The numbers show that reordering is indeed a valuable ingredient of our schedulers. The data also confirms that even without the reordering, the algorithms still notably outperform HDagg, which is the current state-of-the-art synchronous baseline, cf. Table 7.1. Note that the narrow bandwidth random graphs are close to sequential by design, allowing limited parallelization, and thus any valid schedule here already exhibits good locality without reordering; as such, the reordering technique is not beneficial in this case. How to best reorder to improve locality for SpTRSV is research topic that deserves further attention.

Data set	Funnel Locking	Locking	Funnel <i>p</i> -ivotal p.	p-ivotal p.
SuiteSparse	10.70 / 7.86	$9.35 \ / \ 7.36$	$10.44 \ / \ 7.42$	9.27 / 7.03
Erdős–Rényi	14.08 / 8.91	$14.09 \ / \ 8.95$	14.92 / 9.10	15.00 / 9.10
Narrow bandw.	2.79 / 3.19	2.40 / 2.64	3.25 / 3.20	2.53 / 2.69

TABLE 7.4. Geometric mean of speed-ups relative to Serial of our algorithms 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 algorithms over the different processors and architectures in Table 7.5. The data confirms that our algorithms consistently outperform 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	(Funnel) Locking	(Funnel) <i>p</i> -ivotal path	\mathbf{SpMP}	HDagg
Intel x86	10.70 / 9.35	10.44 / 9.27	7.39	3.30
AMD x86	5.99 / 5.40	5.80 / 5.42	4.39	2.12
Huawei ARM	9.30 / 8.89	9.07 / 8.80	n/a	2.03

TABLE 7.5. 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 algorithms scale with a growing number of cores. To examine this, we illustrate the speed-ups (over serial execution) for different numbers of cores in Table 7.6. We note that this experiment was conducted on the AMD x86 machine as it has 64 available cores on a single socket.

Algorithm	2	4	8	16	24	32	40	48	56	64
Funnel Locking	1.86	2.92	3.40	5.24	6.19	6.72	6.92	7.02	6.87	6.65
Funnel p -ivotal path	1.85	2.89	3.31	5.11	6.04	6.45	6.62	6.76	6.63	6.52

TABLE 7.6. Geometric mean of speed-ups relative to Serial of Funnel p-ivotal path and Funnel Locking for different number of cores on the AMD x86 machine taken over the SuiteSparse data set.

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 7.3. This shows that our algorithms do scale if the matrices allow for it and that the number of cores is an important parameter.



FIGURE 7.3. Geometric mean speed-ups of Funnel *p*-ivotal path 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 final 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 ratio of the time it took to compute the schedule to the difference in execution times between the serial execution and the parallel execution using the respective schedule³. 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 7.7 presents the amortization threshold for several scheduling algorithms with the 25th percentile, median, and 75th percentile values shown for each algorithm.

Algorithm	25th percentile	Median	75th percentile
Funnel Locking	740.96	870.26	1158.46
Funnel p -ivotal path	731.91	867.67	1047.27
Locking	83.58	111.36	176.16
<i>p</i> -ivotal path	52.14	63.92	120.73
SpMP	4.91	6.44	8.93
HDagg	489.88	1306.06	2065.96

TABLE 7.7. 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.

A noticeable gap exists between the funneled and un-funneled versions of both the p-ivotal path and Locking algorithms. The funneled versions involve coarsening, a process that is computationally expensive and results in higher scheduling times. However, this added cost is justified when schedules are reused more than 1000 times in magnitude. In such scenarios, the coarsening pays off by reducing execution time, leading to a greater overall benefit in repeated parallel execution performance.

SpMP consistently exhibits low scheduling times across all percentiles, resulting in exceptional amortization threshold. This reflects the strong engineering behind SpMP, which minimizes scheduling overhead compared to the other algorithms. Our scheduling algorithms are research prototypes and single threaded, and as such, their run time could likely be reduced

³If the parallel execution is slower than the serial one, then the amortization threshold is defined as $+\infty$.

with further engineering. For example, the matrices may be split into blocks, c.f. Section 1.2, and our algorithms can be applied to each triangular block in parallel. The resulting block schedules are then easily combined into one schedule for the whole matrix. In any case, in scenarios where schedules are reused often enough, our proposed algorithms in their current implementation already demonstrate significant reduction in overall execution time.

8. Conclusion and future directions

The results show that our barrier list schedulers indeed significantly speed up the parallel SpTRSV kernel, reducing the execution time by a $1.45 \times$ geometric-mean factor compared to SpMP and $3.24 \times$ compared to HDagg on the SuiteSparse benchmark. The data also shows that the three main components of our approach (algorithms, coarsening, and reordering) indeed all contribute to the speed-up, and that the improvement is consistent over various matrix types and architectures.

Future work may consider the adaptation our algorithms to non-uniform memory access (NUMA) architectures. In particular, the AMD x86 data in Section 7.6 confirms that our algorithms scale well to a high number of cores. However, when solving SpTRSV on highly NUMA architectures, we expect our parallel execution schedules to be less effective. It is an interesting question whether one can develop scheduling algorithms that can also efficiently adapt to NUMA, for example, by considering non-uniform bandwidth or latency.

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 merge our algorithms with the block decomposition techniques described in Section 1.2, or to adapt them to a semi-asynchronous setting as in SpMP, in order to allow for a more flexible parallel execution. These methods could allow for further speed-ups on top of our current results.

Appendix A. Tables of matrices

We also include some basic statistics of the matrices used in the experiments, cf. Section 6.2.

Matrix	Dimension	#Non-zeros	Avg. wavefront
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	$32,96\overline{1,525}$	298
shipsec1	140,874	$3,\!977,\!139$	67
StocF-1465	$1,\!465,\!137$	11,235,263	487
thermal 2	$1,2\overline{28,045}$	$4,\!\overline{904,\!179}$	991

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

Matrix	Size	#Non-zeroes	Avg. wavefront
ErdosRenyi_100k_19m_A	100,000	19,999,021	109
ErdosRenyi_100k_19m_B	100,000	19,998,182	109
ErdosRenyi_100k_19m_C	100,000	19,997,897	107
ErdosRenyi_100k_19m_D	100,000	19,995,405	106
ErdosRenyi_100k_19m_E	100,000	19,994,516	107
ErdosRenyi_100k_19m_G	100,000	$19,\!989,\!535$	106
$ErdosRenyi_100k_19m_H$	100,000	$19,\!999,\!989$	110
ErdosRenyi_100k_1m_A	100,000	1,001,528	1,785
ErdosRenyi_100k_1m_B	100,000	1,000,452	1,818
$ErdosRenyi_100k_1m_C$	100,000	1,000,315	1,818
$ErdosRenyi_100k_1m_E$	100,000	1,000,044	1,666
ErdosRenyi_100k_1m_F	100,000	1,000,406	1,785
ErdosRenyi_100k_1m_G	100,000	1,001,171	1,724
ErdosRenyi_100k_1m_H	100,000	1,001,551	1,886
ErdosRenyi_100k_1m_I	100,000	1,000,237	1,639
ErdosRenyi_100k_1m_J	100,000	$1,\!001,\!533$	$1,\!851$
ErdosRenyi_100k_20m_F	100,000	20,001,732	107
ErdosRenyi_100k_20m_I	100,000	20,006,442	109
ErdosRenyi_100k_20m_J	100,000	20,003,479	109
ErdosRenyi_100k_4m_A	100,000	4,998,205	395
$ErdosRenyi_100k_4m_C$	100,000	4,999,271	398
ErdosRenyi_100k_4m_G	100,000	$4,\!999,\!358$	401
ErdosRenyi_100k_4m_J	100,000	4,996,501	414
ErdosRenyi_100k_5m_B	100,000	5,006,107	411
ErdosRenyi_100k_5m_D	100,000	5,001,575	404
$ErdosRenyi_100k_5m_E$	100,000	5,004,251	400
ErdosRenyi_100k_5m_F	100,000	5,002,190	400
ErdosRenyi_100k_5m_H	100,000	5,000,573	409
ErdosRenyi_100k_5m_I	100,000	5,001,846	400
ErdosRenyi_100k_999k_D	100,000	999,915	1,818

TABLE A.2. Matrices and statistics in the Erdős–Rényi data set used for the evaluation. The average wavefront size has been rounded down.

Matrix	Size	#Non-zeroes	Avg. wavefront
RandomBand_p14_b10_100k_146k_A	100,000	146,565	87
RandomBand_p14_b10_100k_146k_B	100,000	146,328	115
RandomBand_p14_b10_100k_146k_D	100,000	146,972	73
RandomBand_p14_b10_100k_146k_F	100,000	146,855	111
$RandomBand_p14_b10_100k_146k_J$	100,000	146,781	105
RandomBand_p14_b10_100k_147k_C	100,000	147,201	61
RandomBand_p14_b10_100k_147k_E	100,000	147,369	73
RandomBand_p14_b10_100k_147k_G	100,000	147,350	132
$RandomBand_p14_b10_100k_147k_H$	100,000	147,412	85
$RandomBand_p14_b10_100k_147k_I$	100,000	147,132	132
$RandomBand_p3_b42_100k_127k_A$	100,000	127,045	46
$RandomBand_p3_b42_100k_127k_B$	100,000	127,019	55
$RandomBand_p3_b42_100k_127k_C$	100,000	127,708	29
$RandomBand_p3_b42_100k_127k_D$	100,000	127,341	45
$RandomBand_p3_b42_100k_127k_E$	100,000	$127,\!569$	67
$RandomBand_p3_b42_100k_127k_F$	100,000	$127,\!137$	47
$RandomBand_p3_b42_100k_127k_G$	100,000	127,774	52
$RandomBand_p3_b42_100k_127k_H$	100,000	127,029	46
$RandomBand_p3_b42_100k_127k_I$	100,000	$127,\!475$	39
$RandomBand_p3_b42_100k_127k_J$	100,000	$127,\!275$	62
$RandomBand_p5_b20_100k_102k_A$	100,000	$102,\!053$	1,298
$RandomBand_p5_b20_100k_102k_B$	100,000	102,621	1,063
$RandomBand_p5_b20_100k_102k_C$	100,000	102,021	1,298
$RandomBand_p5_b20_100k_102k_D$	100,000	$102,\!968$	1,075
$RandomBand_p5_b20_100k_102k_E$	100,000	$102,\!650$	952
$RandomBand_p5_b20_100k_102k_F$	100,000	$102,\!309$	1,162
$RandomBand_p5_b20_100k_102k_H$	100,000	102,324	1,190
$RandomBand_p5_b20_100k_102k_I$	100,000	102,465	1,369
$RandomBand_p5_b20_100k_102k_J$	100,000	102,244	1,010
$RandomBand_p5_b20_100k_103k_G$	100,000	$103,\!152$	892

TABLE A.3. Matrices and statistics in the narrow bandwidth data set used for the evaluation. The average wavefront size has been rounded down.

Matrix	Dimension	#Non-zeros	Avg. wavefront
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.4. Matrices and statistics from SuiteSparse Matrix Collection [DH11] symmetrically permuted using the fill-reducing method 'METIS_NodeND' of [KK98]. The average wavefront size has been rounded down.

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 Transac*tions on Parallel and Distributed Systems, 32(11):2809–2822, 2021.
- [BLPS24] Toni Böhnlein, Benjamin Lozes, 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.
- [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.
- [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.
- [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.