Leveraging One-Sided Communication for Sparse Triangular Solvers *

Nan Ding† Samuel Williams† Yang Liu ‡ Xiaoye S. Li‡

Abstract

In this paper, we implement and evaluate a one-sided communication-based distributed-memory sparse triangular solve (SpTRSV). SpTRSV is used in conjunction with Sparse LU to affect preconditioning in linear solvers. One-sided communication paradigms enjoy higher effective network bandwidth and lower synchronization costs compared to their two-sided counterparts. We use a passive target mode in one-sided communication to implement a synchronization-free task queue to manage the messaging between producer-consumer pairs. Whereas some numerical methods lend themselves to simple performance analysis, the DAG-based computational graph of SpTRSV demands we construct a critical path performance model in order to assess our observed performance relative to machine capabilities. In alignment with our model, our foMPI-based one-sided implementation of SpTRSV reduces communication time by 1.5× to 2.5× and improves SpTRSV solver performance by up to 2.4× compared to the SuperLU_DIST's two-sided MPI implementation running on 64 to 4,096 processes on Cray supercomputers.

1 Introduction

The sparse triangular solve (SpTRSV) is an indispensable task in a wide range of applications from numerical simulation [1] to machine learning [2]. Walking through their long history that originates from the first numerical linear algebra developed by computer pioneers like John von Neumann, we see massive parallel methods and optimization efforts that involve hundreds or even thousands of contributors with immense coding efforts [3–11]. Along with the development process of these efforts, massive performance analysis is conducted by developers to guide the performance optimization.

With more scientific insights derived from software, the demand for ever finer-resolution problems calls for SpTRSV to use ever larger scales of parallelism. While the demands from the application side continue to grow, the memory capacity per node of HPC architectures has barely changed from Titan [12] to Sunway [13] with 32GB memory per node. The newly announced NO.1 supercomputer Summit [14] has 512GB per node, but it's still insufficient to fit the whole application and its intermediate data within one node. Thus, as distributed-memory parallelism has become an essential component in scientific computing, communication has become a critical performance bottleneck for SpTRSV as well as other directed acyclic graph (DAG) based methods [15].

Parallel scalability of SpTRSV is increasingly expensive due to (1) the complex data dependencies and limited parallelism, and (2) high inter-node communication costs. In recent years, substantial efforts have focused on the distributed-memory parallelization of SpTRSV including efforts that focus on analyzing the matrix structure to exploit more parallelism [9]. Unfortunately, such approaches limit (strong) scaling. By restructuring the communication patterns to reduce the latency and balance the communication volume among processors, others have enabled scalibility to 4000 processes [4]. Despite their effectiveness, inter-node communication time still dominates the total SpTRSV time. This makes the performance of parallel SpTRSV far from satisfactory.

As an attractive methodology to improve the communication efficiency, one-sided communication has started to gain popularity since modern interconnects offer remote direct memory access (RDMA) features. RDMA enables a process to directly access memory on remote processes without involvement of the activities at the remote side [16–18]. foMPI [19] is a fast, MPI-3.0 RDMA library interface on Cray systems. It uses distributed-memory application (DMAPP [20]) and XPMEM [21] for fast inter-node and intra-node one-sided communication. The light-weight asynchronous one-sided communication primitives enable a powerful programming model that provides a pathway to efficient DAG execution and accelerator-based exascale solvers thereby opening a new frontier for numerical methods in high-performance distributed computing.

*This research is supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) programs under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory.

†Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA, (nanding|swwilliams@lbl.gov)

‡Scalable Solvers Group, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA, (liuyangzhuan|xsli@lbl.gov)

Aligned with the advances in hardware and communication paradigms, performance modeling of SpTRSV is critical to assess potential performance gains in terms of machine capability. Modeling SpTRSV depends heavily on the structure of a given matrix and the underlying architecture. Whereas the roofline model [22] can effectively bound performance and identify bottlenecks for well-structured, load-balanced codes, it can only provide a very loose bound on performance for codes like SpTRSV. To that end, Wittmann et al. analyzed SpTRSV in the sparse direct solver package PARDISO [23] focusing on data transfers and floating point operations. This served as inputs for the Roofline to establish realistic bandwidth ceilings.

In this paper, we implement and evaluate a one-sided communication-based distributed-memory SpTRSV and construct a critical path performance model in order to assess our observed performance relative to machine capabilities. We **(1)** exploit the higher effective network bandwidth and lower synchronization costs of one-sided communication paradigms, and use a passive target mode in one-sided communication to implement a synchronization-free task queue to manage the messaging between producer-consumer pairs within the DAG. This task queue mechanism can support any DAG execution; **(2)** construct a critical path performance model in order to assess our observed performance relative to machine capabilities and drive future code optimization; **(3)** reduce SpTRSV communication time by up to 2.5× compared to the Cray MPI two-sided implementation through the use of foMPI one-sided communication, and **(4)** integrate our foMPI one-sided implementation of SpTRSV into `SuperLU_DIST` and attain up to 2.4× SpTRSV speedup from a scale of 64 to 4,096 processes on Cray supercomputers.

2 Related Work

Exploring high performance of SpTRSV is becoming ever more crucial in the multi- and many-core era. Most existing parallel triangular solvers focus on optimizing the on-node performance on various parallel architectures [3, 5–8, 10, 11]. Due to the complex data dependencies in SpTRSV, algorithm optimization has been mainly based on the level-set methods and color-set methods for various parallel architectures. Additionally, there is research focused on optimizing the block structure [9], or analyzing nonzero layout and selecting the best sparse kernels by using machine learning and deep learning methods [24–26].

Despite their effectiveness on a single node, we can still see that the high inter-node communication costs associated with these techniques limit their performance. Totoni et al. [27] propose a non-blocked triangular solver using a 1D process layout with improved computation load balance. A 2D cyclic process layout is used to further improve the computation load balance for triangular solves [28, 29], and an asynchronous binary-tree is proposed to reduce the communication latency [4]. Raghavan et al. [30] computes inverse of the triangular matrices as a sequence of matrix factors to replace substitution steps via more efficient distributed matrix-vector multiplications. Venkat et al. [31, 32] developed several techniques that automatically generate wavefront parallelization of sparse matrix computations with faster level-set scheduling being one of the key targets of their research.

Our baseline is a two-sided SPTRSV implementation that uses a supernodal DAG [33] to express the computation dependency. However, whereas previous work leveraged highly-synchronized two-sided MPI communication, we use a passive target mode in one-sided communication to implement a synchronization-free task queue to manage the messaging between producer-consumer pairs within the DAG. To validate the proposed method, we construct a critical path performance model in order to assess our observed performance relative to machine capabilities.

3 Parallel Triangular Solvers

SpTRSV computes a solution vector x for a $n \times n$ linear system $Lx = b$ [1], where L is a lower triangular matrix, and b is a $n \times k$ right-hand side (RHS) matrix or vector ($k = 1$). For a sparse matrix L, the computation of x_i needs some or all of the previous solution rows x_j, $j < i$, depending on the sparsity pattern of the i^{th} row of L. This computation dependency can be precisely expressed by a DAG. In `SuperLU_DIST` [4], a supernodal DAG [33] is used. For a lower triangular matrix L, a supernode is a set of consecutive columns of L with the triangular block just below the diagonal being full, and the same nonzero structure below the triangular block. After a supernode partition is obtained along the columns of L, we apply the same partition row-wise to obtain a 2D block partitioning. The nonzero block pattern defines the supernodal DAG. We assume $b(K)$ and $x(K)$ represent the subvector associated with supernode K. $L(I, K)$ denotes the nonzero submatrix corresponding to supdernodes I and K. Thus, the solution of subvector $x(K)$ can be computed as Eq. (3.1).

$$x(K) = L(K,K)^{-1}\Big(b(K) - \sum_{I=1}^{K-1} L(K,I) \cdot x(I)\Big) \tag{3.1}$$

[1] We use lower triangular matrix to formulate the problem in the paper. Note that the proposed methodology can be easily ported to solve an upper triangular system $Ux = b$.

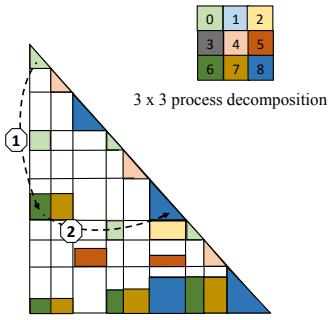


Figure 1: Data flow and process decomposition of `SuperLU_DIST`. The dashed arrows represent the data flow during the solve phase. The numbers represent two communication behaviors: ① block column broadcast, and ② block row reduction.

`SuperLU_DIST` partitions the matrix L among multiple processes using a 2D block cyclic layout. Each process is in charge of a subset of solution subvectors $x(K)$. The solution of these subvectors and partial summation results need communication during the solve phase. Figure 1 describes the data flow of a sparse triangular solve using a 3×3 process decomposition. Processes that assigned to the diagonal blocks (diagonal processes) compute the corresponding blocks of x components. Within one block column, the process owning $x(I)$ sends $x(I)$ to the process of $L(K, I)$ as No. ① in Figure 1. After receiving the required $x(I)$ entries, each process computes its local summation. Within one row, the local sums are sent to the diagonal processes which performs the inversion (No. ② in Figure 1). An asynchronous binary-tree is used to perform the column broadcast and row reduction to reduce the communication latency via `MPI_Isend/MPI_Recv`. The binary tree is built in the setup phase that is executed once for multiple solves. A broadcast tree per supernode column K is built for the processes participating in the column broadcast. Similarly, one reduction tree is built per supernode row I within the processes participating the row reduction. Note that each process in a tree only keeps track of its parent and children.

Table 1: Communication fractions on NERSC's Cori KNL and Cori Haswell as a function of core concurrency.

Matrix	Cori KNL			Cori Haswell		
	256	1024	4096	64	256	1024
A30	46.02%	90.04%	91.98%	55.09%	91.02%	92.34%
chipcool0	36.74%	90.20%	93.33%	44.56%	97.25%	98.34%
shipsec1	45.82%	75.47%	85.18%	40.32%	67.57%	95.56%
gas_sensor	51.00%	87.38%	93.49%	94.59%	95.10%	97.49%
g7jac160	60.04%	67.16%	96.57%	76.09%	85.78%	93.81%
copter2	76.34%	81.96%	88.42%	48.47%	60.06%	89.36%

Table 1 shows that despite all of SuperLU's optimizations, communication still dominates the solve time — at 1024 processes, communication takes over 90% of the time. The communication fraction is calculated as total communication time divided by total solve time.

4 Experiment Setup

Throughout this paper, we conduct the experiments on the Cray XC40 system with two different kinds of nodes: (1)"Cori Haswell" system at NERSC, and (2) "Cori KNL" system at NERSC. Cori Haswell is a Cray XC40 system and consists of 2,388 dual-socket nodes with Intel Xeon E5-2698v3 processors with 16 cores per socket. Each node runs at 2.3 GHz and is equipped with 128GB of DDR4 memory at 2133MHz. The nodes are connected through the Cray Aries Dragonfly interconnect [34]. Cori KNL is on the same Cray Aries high speed inter-node network and consists of 9,688 nodes. Each Cori KNL node is a single-socket Intel Xeon Phi 7250 processor with 68 cores per node running at 1.4 GHz. All experiments in the paper use one thread per MPI process. We use 32 processes per node on Cori Haswell and 64 processes per node on Cori KNL. On both Cori Haswell and Cori KNL, we use the Intel Compiler version 18.0.2 with `-Ofast`, uGNI version 6.0.14, DMAPP 7.1.1, and XPMEM 2.2.15.

Table 2 presents the key facets of the matrices used in this paper. These matrices have also been used in various computational research [35–38]. Matrix A30 is generated from M3D-C1, a fusion simulation code used for magnetohydrodynamics modeling of plasma [36]. All other matrices are publicly available through the SuiteSparse Matrix Collection [39]. The selected matrices cover a wide range of matrix properties (i.e., matrix size, sparsity structure, the number of level-sets and application domain). The matrices are first factorized via `SuperLU_DIST` with METIS ordering for fill-in reduction [40]. The resultant lower triangular matrices are used with the proposed one-sided implementation.

5 Task Queues Using One-sided Communication

For SpTRSV, it is well-known that avoiding synchronization is important in attaining high performance. Moreover, such asynchronous communication patterns require a great deal of effort to manage producer-consumer pairings. The producer-consumer communications require two basic semantics: (1) data transmission and (2) "handshaking". Both semantics are provided by the receiver. However, for many algorithms like SpTRSV, the additional network transactions ("handshaking") from the consumers are not needed. To that end, we exploit one-sided communica-

Table 2: Test matrices. *nnb* is number of nonzero-blocks after factorization. Parallelism=$nnb/levels$.

Matrix	nnz in L	nnb	levels	avg. parallelism	description
chipcool0	4,051,030	22,068	43	513	Model Reduction Problem
A30	6,880,248	11,414	54	211	Magnetohydrodynamics
g7jac160	9,297,606	403,144	3	134,381	Economic Problem
copter2	16,402,818	172,502	93	1,854	Computational Fluid Dynamics Problem
gas_sensor	22,802,879	40,409	59	684	Model Reduction Problem
shipsec1	38,252,451	78,548	8	9,818	Structural Problem

tion to facilitate the overlap of communication and local computation. The lower overhead associated with one-sided communication compared to the traditional two-sided communication has the potential to increase the performance at scale by increasing the effective network bandwidth and reducing synchronization overheads.

One-sided MPI offers two synchronization modes: the active target mode and passive target mode [41]. The active mode requires four MPI operations per message: `MPI_Win_post` and `MPI_Win_wait` from the consumer, `MPI_Win_start` and `MPI_Win_complete` from the producer. In the passive target mode, only producers are involved in message transfers. There are two kinds of passive target mode: using locks or no lock. Using locks means that other processes can not access the exposed RMA buffer before unlock. No lock means that all processes can access the RMA buffer at any time. As a consequence, the active target mode has at least four times as many MPI operations as the passive target mode. Furthermore, in the passive target mode, using locks has at least twice as many MPI operations as no lock.

Due to the frequent and asynchronous messages in one solve, we use a passive target mode with no lock (`MPI_Win_lock_all` and `MPI_Win_unlock_all`) at the beginning and ending of one solve for each block column and each block row. All the enqueue operations and dequeue operations in one block column (or block row) are performed within a pair of `MPI_Win_lock_all` and `MPI_Win_unlock_all`. Therefore, we avoid the extra synchronization overhead compared to both passive target mode using locks and the active target mode.

For each message, we focus on two kinds of one-sided communication: `MPI_Put` and `MPI_Accumulate` with `MPI_Sum`. These two operations are supported by both the Cray MPI [42, 43] and foMPI [19] implementations. Each task queue is allocated once using `MPI_Win_create` which exposes the local memory to RMA operations by other processes in a communicator. The resultant task queue is reused on multiple solves.

Figure 2 shows the computational structure for solving a lower triangular system when using a 3×3 process decomposition. Process numbers are marked with colors, and the number in each circle is the block number. The dashed arrows represent the enqueue operation among different processes. Recall that there are two kinds of communication in SpTRSV: block column broadcasts and block row reductions. Processes participating in the block column broadcast send the corresponding components of the solution vector (x_i) to their children. Processes participating in the block row reduction send the corresponding partial sum ($lsum_i$) to the diagonal process. Reflecting these two communication patterns, we create two task queues for each process: a broadcast task queue (BTq) and a reduction task queue (RTq) to enqueue the ready tasks from block column broadcasts and block row reductions. We allocate the task queues of each process before entering the solving phase bounding the size of the task queue (TqS) according to the block size and process decomposition as Eq. (5.2) shows. The TqS is a function of the number of processes along block columns/rows (P_x/P_y) and the message count of the processes in each block column/row. In Eq. (5.2), C_i is the received number of tasks from process i and mz is the maximum size of an individual message.

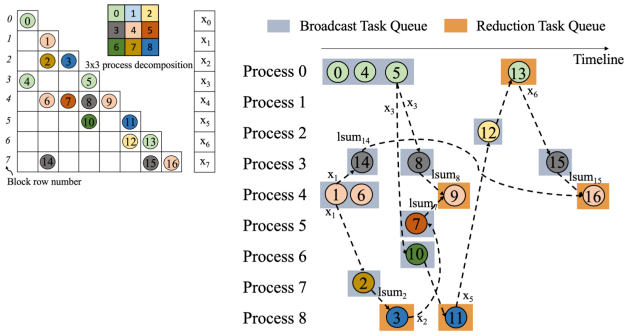


Figure 2: The computational structure for solving a lower triangular system. The dashed arrows represents the enqueue operation among different processes. Process rank is denoted by color, while the number in each circle is the block number.

$$TqS = \begin{cases} \sum_{i=0}^{P_y} C_i \cdot mz, & \text{if BTq} \\ \sum_{i=0}^{P_x} C_i \cdot mz, & \text{if RTq} \end{cases} \tag{5.2}$$

Let us consider how BTq and RTq operate in the context of Figure 2. We commence solving the lower triangular system with the diagonal blocks (block 0, 1 in Figure 2) that have no non-zero blocks within the same block row. We then enqueue the corresponding x components to the BTq of its children, e.g. $1 \rightarrow 2$ and $1 \rightarrow 14$. When it comes to row reductions, we enqueue the corresponding partial sum $lsum$ to the diagonal process within the same block row, e.g. $2 \rightarrow 3$, $7 \rightarrow 9$ and $8 \rightarrow 9$. Once each process fetches a new task from the task queue, the process will check whether it has children in the binary communication tree that was built in the setup phase. If it has children in BTq, the process will send the x components first and then begin local computation, e.g. process 0 (owns block 5) will enqueue its x components to process 3 (owns block 8) and process 6 (owns block 10), and then begins to compute block 5. When a process has more than one task in the BTq such as process 4 in Figure 2, process 4 will compute the leaf node block 1 first, and then send corresponding x components to process 7 (owns block 2). After that, process 4 begins to compute block 6. If it has children in the RTq, the process will send the corresponding partial sum $lsum$ to its parent after the local computation completes.

Without two-sided MPI's "handshaking" between the producer-consumer pairs, we provide two methods, *EnQueue_payload* and *EnQueue_counter*, for the consumers to ensure data arrival completion. Figure 3 illustrates the task queue design for block column broadcasts using a 3×3 process decomposition as an example. We split the global communicator into three sub-communicators. In each sub-communicator, processes will perform broadcasts within block columns with each process maintaining a broadcast task queue. The tasks from different producers have different offsets in the task queue. We compute these offsets according to the dependency graph in the setup phase. Reduction task queues within block rows use the same idea.

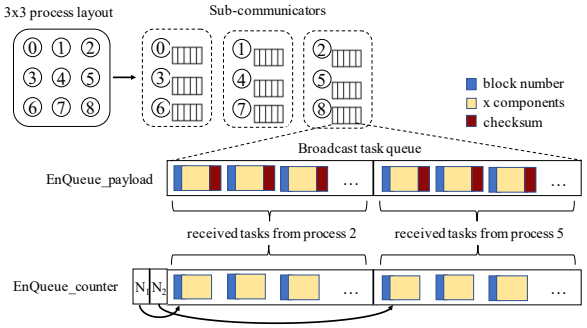


Figure 3: One-sided implementation of task queues. N_i in EnQueue_counter represents the number of received tasks from process i. The size of BTq using EnQueue_payload is $\sum C_i$ words larger than the two-sided implementation. The size of RTq using EnQueue_counter is $\sqrt{P}$ words larger than the two-sided implementation.

In the case of *EnQueue_counter*, we keep counters for each producer. The value of the counter N_i indicates the total number of enqueued tasks for process i as Figure 3 shows. The size of the task queues using *EnQueue_counter* (Eq. (5.3)) needs an additional $\sqrt{P}$ words for the counter compared to Eq. (5.2) when using a 2D process decomposition. Once the producer sends a message, it then performs an atomic read-modify-write operation that is implemented by `MPI_Accumulate` with `MPI_Sum`. This atomic read-modify-write operation increases the counter of the corresponding task queue by one each time. We also track the number of dequeued tasks. Thus, the difference of these two values is the number of tasks that can be solved in the current queue. When N_i equals to C_i, it indicates that the consumer has received all tasks from process i.

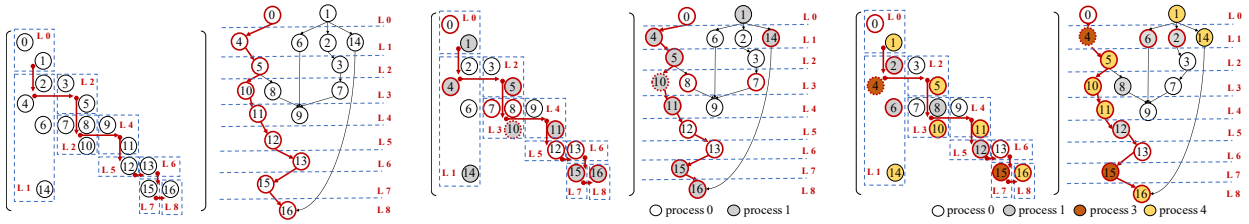
$$TqS = \begin{cases} \sum_{i=0}^{P_y} (P_y + C_i \cdot mz), & \text{if BTq} \\ \sum_{i=0}^{P_x} (P_x + C_i \cdot mz), & \text{if RTq} \end{cases} \tag{5.3}$$

In the case of *EnQueue_payload*, since each individual message may be split into packets for out-of-order transmission and certainly not delivered atomically, we employ a checksum [44] to determine when the data for each individual message has completely arrived. The checksum is a function of the summation of the block number and the corresponding components of x (or partial summation) in each message in BTq (or RTq). Once a message arrives, the consumer will conduct a checksum for the received task. If the checksum equals the payload at the end of this received task, the consumer deems that all data has arrived. The consumer spins until the checksum succeeds (data from the sender is RDMA'd). The TqS for the *EnQueue_payload* (Eq. (5.4)) needs an extra $\sum C_i$ words compared to Eq. (5.2).

$$TqS = \begin{cases} \sum_{i=0}^{P_y} C_i \cdot (mz + 1), & \text{if BTq} \\ \sum_{i=0}^{P_x} C_i \cdot (mz + 1), & \text{if RTq} \end{cases} \tag{5.4}$$

6 SpTRSV Performance Model

We constructed a performance model (Algorithm 1) in order to understand SpTRSV performance and quantify the potential performance benefits without running the SpTRSV code. The challenge lies in how to abstract



(a) The initial critical path is the path that goes through every level: $0 \rightarrow 4 \rightarrow 5 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 15 \rightarrow 16$.

(b) The refined critical path using 2×1 processes: $0 \rightarrow 4$, $14 \rightarrow 5 \rightarrow 7$, $8 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 15 \rightarrow 16$.

(c) The refined critical path using 2×2 processes: $0 \rightarrow 2$, $6 \rightarrow 5 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 15 \rightarrow 16$.

Figure 4: The initial critical path based on the level-set method using BFS and the refined critical paths assuming a 2×1 and 2×2 process decomposition. The dashed red circle is the block that is replaced by the solid red circles within the same level when refine the critical path according to the process decomposition. Li represents level i.

the code behavior to characterize the performance features for such a complicated asynchronous code. To address this, we first conduct a critical path analysis. We then model the computation time and communication time separately based on the critical path.

Critical path analysis: The critical path analysis follows the task dependency graph of the sparse matrix based on the well-known level-set method [45, 46] using a breadth-first search [47]. The critical path analysis has two phases: (1) find the initial critical path according to the task dependencies, and (2) refine the critical path in (1) according to the process decomposition.

Figure 4a shows the initial critical path of an 8×8 sparse lower triangular matrix based on the level-set method. The numbers represent the non-zero block number. We first find the task dependency of the matrix, and then group the blocks that can be solved in parallel. Thus, blocks in the same level (Li represents level i in Figure 4a) can be solved concurrently. In this way, the initial critical path of the matrix is the path that goes through every level. The initial critical path is $0 \rightarrow 4 \rightarrow 5 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 15 \rightarrow 16$.

The next step in our performance modeling is to refine the critical path according to the process decomposition. The refined critical path could become longer due to limited resources. When a process owns multiple blocks in a level, timing is adjusted to reflect intra-process serialized execution of blocks as Figure 4b shows. Here we use the timing from the process who has the maximum number of blocks to represent the computation run time within a level. For example, the time cost of level 1 (L1 in Figure 4b) depends on block 4 and block 14 since these two blocks need to be executed sequentially in process 0. The time cost of level 3 depends on block 7 and block 8 since these two blocks are assigned to process 1 while process 0 has only one block. So the refined critical path using a 2×1 process decomposition has 11 blocks: $0 \rightarrow 4$, $14 \rightarrow 5 \rightarrow 7$, $8 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 15 \rightarrow 16$. Similarly, in Figure 4c, we use block 2 and block 6 (from process 1) to replace block 4 from process 3 in the initial critical path. Thus, the refined critical path using a 2×2 process decomposition (10 blocks) is $0 \rightarrow 2$, $6 \rightarrow 5 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 15 \rightarrow 16$. With more processes being used, the length of the refined critical path using a 2×2 process decomposition is reduced by one compared to the refined critical path using a 2×1 process decomposition. Following this idea, we conduct the critical path analysis according to the matrix pattern and critical path refinement according to the process decomposition.

Modeling computation: We modeled the computation time using a lower bound (LB, line 4 in Algorithm 1) and an upper bound (UB, line 6 in Algorithm 1). The real run time lies in between this range. The closer to the lower bound, the better the task placement the current code achieves. Both UB and LB are bandwidth-constrained models. We use a fixed peak memory bandwidth per process (BW_{mem}) in the model to estimate the maximum number of processes the code can be scaled to. For each block, its computation time is modeled using total data movement (Bytes) divided by the BW_{mem}. The UB is modeled by accumulating the computation time of blocks on the critical path. Note that the LB is the maximum time among all processes. The LB is never a better estimate compared to the perfect parallelization because perfect parallelization is a simple, highly-optimistic linear scaling estimate from a sequential implementation that ignores any potential load imbalance, communications, or critical path dependencies.

Modeling communication: The communication time of a lower triangular matrix is modeled using the in-degree ($\#in$) and out-degree ($\#out$) of the diagonal blocks ($\#row = \#col$) on the critical path (line 7- 13 in Algorithm 1). Here we count the $\#in$ and $\#out$ according to the process decomposition. No communication is needed if the blocks are assigned to the same process with the diagonal block. The $\#out$ refers to the number of messages this diagonal block will send to the others when performing block column broadcast. Each message has a size of the width of block column i (w_i). The latency of multiple sends can be overlapped because all the messages are coming from the same producer. Let's assume there are P processes partic-

Algorithm 1 Performance modeling SpTRSV

Model inputs: w_i: width of block i
h_i: height of block i
BW_{mem}: peak memory bandwidth per process (words/second)
BW_{net}: inter-node network bandwidth (words/second)
L_{net}: inter-node network latency
P: number of processes
Model outputs: LB: Computation lower bound
UB: Computation upper bound
$COMM$: Communication time

1: **procedure** MODELING
2: Analyze the critical path C
3: Count $\#in$ and $\#out$ of diagonal blocks
4: $LB = \max(\sum \frac{w_i \cdot h_i}{BW_{mem}} \mid block_i \in P_i), 0 \leq i < P$
5: **while** $block(\#row, \#col) \in C$ **do**
6: $UB \mathrel{+}= \frac{w_i \cdot h_i}{BW_{mem}}$
7: **if** $block(\#row, \#col)$ is a diagonal block **then**
8: **if** Binary-tree **then**
9: $COMM \mathrel{+}= \log_2(\#in) \cdot (L_{net} + \frac{roundup(h_i,2)}{BW_{net}}) + L_{net} + \frac{\log_2(\#out) \cdot roundup(w_i,2)}{BW_{net}}$
10: **else** ▷ Flat-tree
11: $COMM \mathrel{+}= \#in \cdot (L_{net} + \frac{roundup(h_i,2)}{BW_{net}}) + L_{net} + \frac{\#out \cdot roundup(w_i,2)}{BW_{net}}$
12: **end if**
13: **end if**
14: **end while**
15: **end procedure**

ipating in the block column broadcast. When using a binary communication tree, each process that participates in the block column broadcast sends at most two messages to its children instead of the $P-1$ messages associated with a flat communication tree. This reduces the send message count of the corresponding process from $P-1$ to $log_2 P$. The $\#in$ represents the number of messages this diagonal block should receive to complete its local computation when performing the block row reduction. Each message size equals the height of block rows i (h_i). The parent nodes in the reduction tree must serialize incoming data from their children. This pattern causes congestion. Each process participating in the block row reduction receives two messages instead of $P-1$ messages when using a binary communication tree. Note that the latency difference between the binary tree and the flat tree is reflected at line 9 and line 11 of Algorithm 1. We use a round-trip ping pong benchmark to characterize the network parameters inter-node network bandwidth (BW_{net}) and inter-node network latency (L_{net}) for each communication runtime. BW_{net} and L_{net} are parameterized by benchmarked message sizes. $roundup(x, 2)$ rounds the message size x to the next power of two to match the corresponding BW_{net} in the model.

7 Performance and Analysis

In this section, we first illustrate how to use the proposed performance model to conduct performance analysis including the computation/communication time and the performance impact with different process decompositions. We then show the performance improvement of our one-sided implementation on Cori KNL and Cori Haswell.

7.1 Understanding the run time.

Model-based performance analysis is critical to assess potential performance gains in terms of machine capability. In this section, we first demonstrate how we extract the model parameters. In order to affect a concise, yet detailed performance analysis of computation and communication time, we visualize the results of our critical path performance model and apply it to two representative matrices: A30 and g7jac160. The two selected matrices have very different features (Table 2): e.g., number of non-zero blocks, number of levels, and number of parallelisms, which can cover a wide range of matrix properties.

Extracting model parameters: We evaluate four different implementations of SpTRSV:

- Cray's two-sided MPI: MPI_Isend and MPI_Recv
- Cray's one-sided MPI: MPI_Put
- foMPI (Enqueue_payload): foMPI_Put
- foMPI (Enqueue_counter): foMPI_Accumulate with foMPI_Sum

Note, Cray's two-sided MPI implementation is highly-optimized for the Aries network, is considered a gold-standard, and is the default at NERSC. It thus provides an exceptionally challenging baseline and forms the basis for our comparisons to foMPI in the rest of the paper.

The network parameters (BW_{net} and L_{net}) integral to our performance model are measured using a round-trip ping-pong within any two of the total processes. Figure 5 shows the results on Cori KNL using 16 nodes. The BW_{net} and L_{net} used in our model are averaged across multiple ping-pongs among all pairs of multiple nodes. The foMPI(Enqueue_payload) implementation achieves up to 3× and 8× speedup compared to the Cray MPI two-sided and Cray MPI one-sided implementations respectively.

The implementation with foMPI (Enqueue_counter) does not achieve better performance due to (1) the doubling of message counts (for each message the producer sends one message containing data and the other message with a counter), and (2) the reduced performance of the atomic read-modify-write operation via foMPI_Accumulate together with foMPI_Sum.

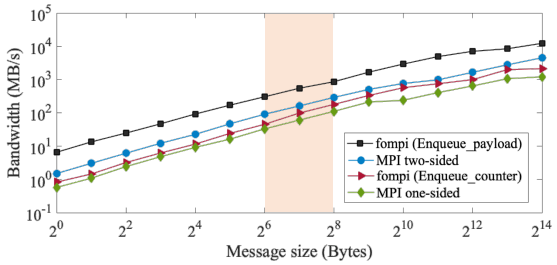


Figure 5: Microbenchmark showing BW_{net} on 16 Cori KNL nodes. The tan stripe highlights the typical message size in SpTRSV of 64 bytes to 256 bytes. Observe the substantial potential of fompi over two-sided MPI for these sizes.

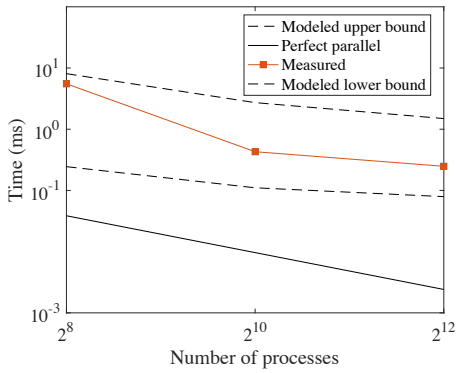


Figure 6: Computation upper and lower bounds for matrix A30. The gap between the modeled lower bound and the measured data indicates the potential performance benefit from further optimizations that improve task placement.

Computation: Figure 6 shows the modeled upper bound and the lower bound of matrix A30. The measured data is close to the upper bound at first and then the gap with the upper bound becomes large. This reflects the fact that our implementation assigns blocks with smaller block row number higher execution priority. Thus, at low concurrency, each process has less idle time. This optimization will not shorten the critical path but will reduce the dependencies from critical path refinement, and will improve load balance. The gap between the modeled lower bound and the measured data indicates the potential performance benefit from further optimization that improve task placement. We can see there is a large gap between the modeled lower bound and perfect parallelization. Recall that perfect parallelization is a simple, highly-optimistic linear scaling estimate from a sequential implementation that ignores any potential load imbalance, communication, or critical path dependencies. As such, our model's lower bound provides more realistic best case than the unrealistic perfect parallelization.

Communication: Table 3 shows the modeling results of matrix A30 and matrix g7jac160 using three different implementations: (1) Cray's two-sided MPI, (2) foMPI (Enqueue_payload), and (3) foMPI (Enqueue_counter). The speedup is calculated by using the two-sided modeled/measured time divided by modeled/measured time of the other two implementations. The measured speedup of foMPI (Enqueue_payload) is less than the modeled ones due to the overhead to check the data complete arrival.

As we discussed in Algorithm 1, we can determine that a smaller number of dependent diagonal blocks on the critical path leads to less communication time. Figure 7 is a visualization of the critical path of matrix A30 using 256(left) and 4096(right) processes. In essence, this is a sparsity plot of the matrix wherein grey points represent non-zero blocks, the blue line is the critical path using 256 processes, and dependent blocks are marked with orange dots according

Table 3: Communication time (ms) of matrix A30 and g7jac160 on Cori KNL as a function of core concurrency and three implementations. Note that implementation of foMPI Enqueue counter has a low performance due to the atomic read-modify-write operation via MPI_Accumulate with MPI_Sum. Speedup (S) equals to the two-sided modeled/measured time divided by modeled (M)/measured (R) time of the other two implementations.

		A30			g7jac160		
		256	1024	4096	256	1024	4096
Cray MPI	M	9.9	8.1	6.9	43.7	27.0	12.0
two-sided	R	9.2	7.1	6.2	36.6	28.0	10.9
foMPI	M	3.6	2.9	2.1	21.6	13.1	7.7
one-sided	R	4.2	3.4	2.6	21.5	15.9	6.8
with	S_M	**2.7**×	**2.7**×	**3.2**×	**2.0**×	**2.0**×	**1.6**×
payload	S_R	**2.2**×	**2.0**×	**2.4**×	**1.7**×	**1.8**×	**1.6**×
foMPI	M	67.6	52.2	41.0	221.8	202.9	112.5
one-sided	R	71.7	46.3	37.8	251.8	233.6	130.4
with	S_M	**0.15**×	**0.16**×	**0.16**×	**0.19**×	**0.13**×	**0.10**×
counter	S_R	**0.13**×	**0.15**×	**0.16**×	**0.15**×	**0.12**×	**0.08**×

to the critical path refinement. Thus, all diagonal blocks (*#in* and *#out*) on the critical path contribute to the total communication time. When comparing Figure 7(left) and (right), both the number of off-diagonal blocks and diagonal blocks decrease as the number of processes increases. This indicates both computation time and communication time scale. We can use the model to estimate how total solve time scales. Assume that we use a 2D square process decomposition with three parallel decompositions: 8×8, 10×10, and 12×12. The corresponding numbers of blocks as observed on their critical paths are 192, 115, and 85 respectively. That is to say the total solve time can scale with $\sqrt{P}$. Moreover, the number of diagonal blocks are 125, 77, and 82 respectively which indicates that reductions in run time are derived from both improvements in communication computation from 256 to 1024 processes. However, from 1024 to 4096 processes, communication time increases. Nevertheless, there is still a performance benefit from 1024 to 4096 processes due to reductions in computation time.

7.2 Understanding the process decomposition.

An optimal process decomposition can achieve a faster solve time. In this section, we show how performance differentiates across varying process decompositions using two matrices with extreme sparsity patterns: a dense lower triangular matrix and a bi-diagonal lower triangular matrix. Note that the optimal process decomposition for a general sparse matrix depends heavily on the matrix size, pattern, and the architecture. We consider 2D square and 1D process decompositions here, but other irregular or adaptive decompositions can also be analyzed with the proposed performance model.

For the dense lower triangular matrix (Figure 8), we can still see the computation time and total solve time scale while the communication time does not. As for the bi-diagonal

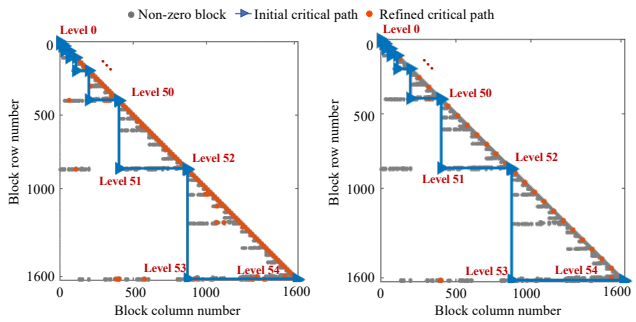


Figure 7: Critical path visualization of matrix A30 with 256(left) and 4,096 processes(right). Fewer off-diagonal blocks improves computation time. Fewer diagonal blocks improves communication time.

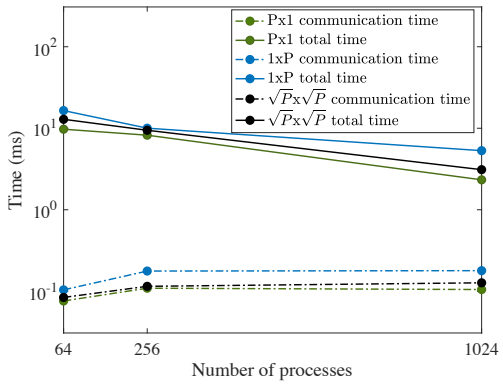
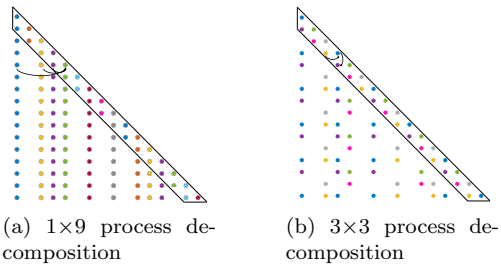


Figure 8: Solid vs. dashed are for total solve time and communication time of the dense lower triangular matrix with different process decomposition. The communication time does not scale because all the diagonal blocks and sub-diagonal blocks are on the critical path. The total solve time scales due to the increasing computation parallelism from the off-diagonal blocks.



(a) 1×9 process decomposition

(b) 3×3 process decomposition

Figure 9: The sparsity plot as observed on the critical path of the dense matrix wherein process numbers are marked with colors. Less off-diagonal blocks indicts a less computation run time.

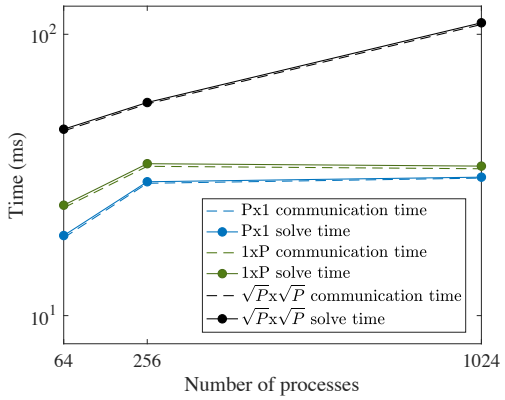
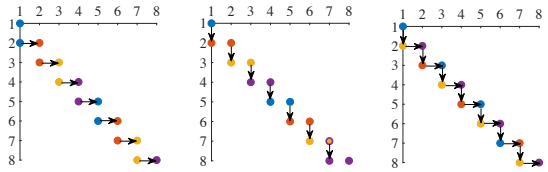


Figure 10: Solid vs. dashed are for total solve time and communication time of the bi-diagonal matrix with different process decomposition. Both the total time and the communication time do not scale since all blocks of the bi-diagonal matrix are on the critical path.

lower triangular matrix (Figure 10), both computation time and communication time do not scale. We use $m \times n$ to represent the 2D process decomposition, where m is the number of processes in each block column. The dense lower triangular matrix is of size 1200×1200 stored in sparse format where each non-zero block is of size 1×1. Figure 8 shows the scalability as a function of process decomposition. The communication time does not scale because all the diagonal blocks and sub-diagonal blocks are on the critical path. The total solve time scales with P because there is still ample parallelism among the off-diagonal blocks. That is to say, reductions in solve time are derived solely from scaling computation time.

We illustrate these effects by examining a small (16×16 where each non-zero block is of size 1×1) dense matrix. Figure 9a shows the critical path of the small dense matrix with different process decompositions. We can see that all the diagonal blocks and and sub-diagonal blocks are on the critical path regardless of process decomposition. The communication time using the 1×P process decomposition is larger than the P×1 process decomposition and the $\sqrt{P} \times \sqrt{P}$ process decomposition because the block row reduction for the diagonal processes serializes all incoming data. The total $\#in$ of all diagonal blocks using the 1×9 process decomposition is 2.2× larger than the one using the 1×4 process decomposition. Similarly, the P×1 process decomposition will not scale either. The difference is that the total network latency of the 1×P process decomposition is P× larger than the P×1 process decomposition. So the performance of the P×1 process decomposition is better than the 1×P process decomposition. The 3×3 process decomposition in Figure 9b shows fewer dependencies of off-diagonal blocks on the critical path, but its row reduction cost makes it slower than the P×1 process decomposition.

Figure 10 shows the scaling performance of a bi-diagonal matrix of 1000×1000 non-zeros where each non-zero block is of size 1×1. The communication time does not scale because all blocks of the matrix are on the critical path. Unlike the dense lower triangular matrix, the total solve time for the



(a) 1×4 (b) 4×1 (c) 2×2

Figure 11: The critical path of the bi-diagonal matrix with different process decomposition. Different processes are marked with different colors. The message count of 1×P and P×1 equals the number of block column/row. The message count of $\sqrt{P}\times\sqrt{P}$ equals to the sum of the number of block columns and the number of block rows.

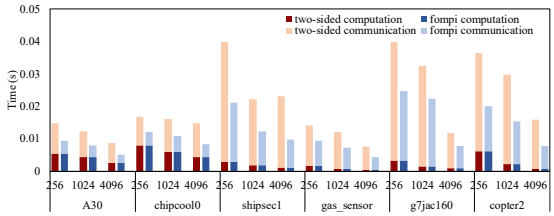


Figure 12: Lower triangular solve time on Cori KNL using foMPI (Enqueue_payload) compared to two-sided MPI. Our SpTRSV implementation reduces communication time by 1.5× to 2.5× and attains a speedup of up to 2.4× on 256 to 4,096.

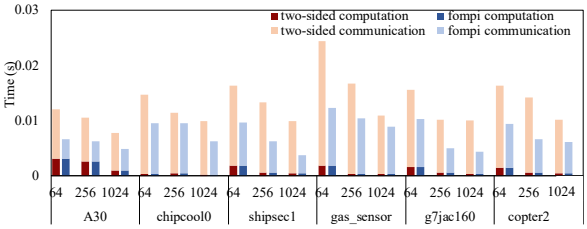


Figure 13: Lower triangular solve time on Cori Haswell using foMPI (Enqueue_payload) compared to two-sided MPI. Our one-sided implementation reduces communication time by 1.5× to 2.5× and attains a speedup of up to 2.3× on 64 to 1,024 processes.

matrix does not scale either since there are no off-diagonal blocks in the matrix. That is, all the blocks in matrix have to be executed sequentially. The performance of the $\sqrt{P}\times\sqrt{P}$ process decomposition is worse than the other two since the message count of the $\sqrt{P}\times\sqrt{P}$ process decomposition equals the sum of the number of block columns and the number of block rows. The message counts of the 1×P process decomposition and the P×1 process decomposition equal the number of block columns and block rows, respectively.

Figure 11 presents the critical path of a small bi-diagonal (8×8 blocks). Once again, the critical path using different process decomposition is the same as the original matrix. In the example in Figure 11, the total message count of the $\sqrt{P}\times\sqrt{P}$ process decomposition is 2× larger than the one using 1×P (or P×1) process decomposition. To be more specific, the communication time with the $\sqrt{P}\times\sqrt{P}$ process decomposition equals to the sum of the communication time using 1×P process decomposition plus the communication time using P×1 process decomposition.

7.3 Overall Performance. Figure 12 shows the overall performance on Cori KNL. Our one-sided SpTRSV implementation using foMPI (Enqueue_payload) reduces communication time by 1.5× to 2.5× and attains a speedup of up to 2.4× on 256 to 4,096 processes compared against SuperLU_DIST's two-sided implementation. Recall that microbenchmarks showed that Cray's MPI one-sided is 8× slower than foMPI (and thus slower than Cray's high-quality two-sided implementation). As such, we do not include Cray's MPI one-sided in this discussion. Figure 13 presents performance on Cori Haswell where our one-sided SpTRSV implementation reduces communication time by 1.5× to 2.5× and improves SpTRSV performance by up to 2.3× on 64 to 1024 processes. Note, the communication time in the figures contains both the one-sided MPI time and the checksum time used to verify data arrival. The total run time speedup is lower than the speedup from the model due to the computation cost. The one-sided implementation of task queues outperforms the two-sided producer-consumer implementation without too much coding effort (3,500 LOCs at software level).

8 Conclusions and Future Work

The low arithmetic intensity, complex data dependencies, and high inter-node communication present immense performance impediments for SpTRSV at high concurrency. In this work, we use foMPI one-sided communication to implement a synchronization-free task queue to manage the messaging between producer-consumer pairs in SpTRSV. We construct a critical path performance model in order to assess our observed performance relative to machine capabilities. The performance model takes the matrix sparsity, process layouts, network bandwidth/latency and memory bandwidth into account to validate the SpTRSV solve time. In alignment with our model, from a scale of 64 to 4,096 processes, our foMPI-based one-sided implementation of SpTRSV reduces communication time by 1.5× to 2.5× and improves SpTRSV performance by up to 2.4× when compared to SuperLU_DIST's implementation using Cray's optimized two-sided MPI.

It is worth noting that even though our work is conducted on manycore architectures, we believe that our proposed method can bring insightful experience for DAG-based computations on emerging accelerated architectures. To that end, in the future, we will extend our one-sided task queue methodology and evaluate other one-sided PGAS models like OpenSHMEM [17]. We will assess the efficacy

of generalizing and encapsulating our work into a one-sided task queue library to facilitate high-performance distributed memory DAG computations.

References

[1] István Z Reguly, Gihan R Mudalige, Carlo Bertolli, Michael B Giles, Adam Betts, Paul HJ Kelly, and David Radford. Acceleration of a full-scale industrial CFD application with OP2. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1265–1278, 2016.

[2] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on uncertainty in artificial intelligence (UAI)*, pages 340–349, 2010.

[3] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. swSpTRSV: a fast sparse triangular solve with sparse level tile layout on Sunway architectures. In *ACM SIGPLAN Notices*, volume 53, pages 338–353. ACM, 2018.

[4] Yang Liu, Mathias Jacquelin, Pieter Ghysels, and Xiaoye S Li. Highly scalable distributed-memory sparse triangular solution algorithms. In *Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pages 87–96. SIAM, 2018.

[5] Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience*, 29(21):e4244–n/a, 2017. ISSN 1532-0634. doi: 10.1002/cpe.4244. URL `http://dx.doi.org/10.1002/cpe.4244`. e4244 cpe.4244.

[6] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*, pages 617–630. Springer, 2016.

[7] Andrew M. Bradley. A hybrid multithreaded direct sparse triangular solver. In *Proceedings of SIAM Workshop on Combinatorial Scientific Computing*, pages 13–22, 2016. doi: 10.1137/1.9781611974690.ch2. URL `http://epubs.siam.org/doi/abs/10.1137/1.9781611974690.ch2`.

[8] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *International Supercomputing Conference*, pages 124–140. Springer, 2014.

[9] Ehsan Totoni, Michael T Heath, and Laxmikant V Kale. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Computing*, 40(9): 454–470, 2014.

[10] Ruipeng Li and Yousef Saad. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[11] Jan Mayer. Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing*, 86(4):291, Sep 2009. ISSN 1436-5057. doi: 10.1007/s00607-009-0066-3. URL `https://doi.org/10.1007/s00607-009-0066-3`.

[12] Oak Ridge Leadership Computing Facility. `https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/`.

[13] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[14] Jonathon Anderson, Patrick J Burns, Daniel Milroy, Peter Ruprecht, Thomas Hauser, and Howard Jay Siegel. Deploying RMACC summit: an HPC resource for the Rocky Mountain Region. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, page 8. ACM, 2017.

[15] Simon Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 154–159. IEEE, 1994.

[16] Robert Alverson, Duncan Roweth, and Larry Kaplan. The gemini system interconnect. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 83–87. IEEE, 2010.

[17] Jeff R Hammond, Sayan Ghosh, and Barbara M Chapman. Implementing OpenSHMEM using MPI-3 one-sided communication. In *Workshop on OpenSHMEM and Related Technologies*, pages 44–58. Springer, 2014.

[18] Sreeram Potluri, Nathan Luehr, and Nikolay Sakharnykh. Simplifying Multi-GPU Communication with NVSHMEM. In *GPU Technology Conference*, 2016.

[19] Roberto Belli and Torsten Hoefler. Notified access: Extending remote memory access programming models for producer-consumer synchronization. In *IEEE International Parallel and Distributed Processing Symposium*, pages 871–881. IEEE, 2015.

[20] Monika ten Bruggencate and Duncan Roweth. Dmapp–an API for one-sided program models on baker systems. In *Cray User Group Conference*, 2010.

[21] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI® AltixTM 3000 global shared-memory architecture. *Silicon Graphics, Inc*, 2005.

[22] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[23] Markus Wittmann, Georg Hager, Radim Janalik, Martin Lanser, Axel Klawonn, Oliver Rheinbach, Olaf Schenk, and Gerhard Wellein. Multicore performance engineering of sparse triangular solves using a modified roofline model. In *The 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 233–241. IEEE, 2018.

[24] Guangming Tan, Junhong Liu, and Jiajia Li. Design and implementation of adaptive spmv library for multicore and many-core architecture. *ACM Transactions on Mathematical Software (TOMS)*, 44(4):46, 2018.

[25] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the gap between deep learning and sparse matrix format selection. In *ACM SIGPLAN Notices*, volume 53, pages 94–108. ACM, 2018.

[26] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, volume 48, pages 117–126. ACM, 2013.

[27] Ehsan Totoni, Michael T. Heath, and Laxmikant V. Kale. Structure-adaptive parallel solution of sparse triangular linear systems. *Parallel Computing*, 40(9):454 – 470, 2014. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2014.06.006. URL `http://www.sciencedirect.com/science/article/pii/S0167819114000799`.

[28] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–17, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-89791-984-X. URL `http://dl.acm.org/citation.cfm?id=509058.509092`.

[29] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003. ISSN 0098-3500. doi: 10.1145/779359.779361. URL `http://doi.acm.org/10.1145/779359.779361`.

[30] Padma Raghavan. Efficient parallel sparse triangular solution using selective inversion. *Parallel Processing Letters*, 8(01):29–40, 1998.

[31] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 41. IEEE Press, 2016.

[32] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Notices*, volume 50, pages 521–532. ACM, 2015.

[33] James W Demmel, Stanley C Eisenstat, John R Gilbert, Xiaoye S Li, and Joseph WH Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[34] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88, June 2008. doi: 10.1109/ISCA.2008.19.

[35] Ulrike Baur, Peter Benner, Andreas Greiner, Jan G Korvink, Jan Lienemann, and Christian Moosmann. Parameter preserving model order reduction for MEMS applications. *Mathematical and Computer Modelling of Dynamical Systems*, 17(4):297–317, 2011.

[36] S C Jardin, N Ferraro, X Luo, J Chen, J Breslau, K E Jansen, and M S Shephard. The M3D-C1 approach to simulating 3D 2-fluid magnetohydrodynamics in magnetic fusion experiments. *J. Phys. Conf. Ser.*, 125(1):012044, 2008. URL `http://stacks.iop.org/1742-6596/125/i=1/a=012044`.

[37] Alexander Ludwig. The Gauss–Seidel–quasi-Newton method: A hybrid algorithm for solving dynamic economic models. *Journal of Economic Dynamics and Control*, 31(5):1610–1632, 2007.

[38] Gary Kumfert and Alex Pothen. Two improved algorithms for envelope and wavefront reduction. *BIT Numerical Mathematics*, 37(3):559–590, 1997.

[39] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL `http://doi.acm.org/10.1145/2049662.2049663`.

[40] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. doi: 10.1137/S1064827595287997. URL `https://doi.org/10.1137/S1064827595287997`.

[41] MPI: A Message-Passing Interface Standard Version 3.1. `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[42] K Kandalla, David Knaak, K McMahon, N Radcliffe, and M Pagel. Optimizing Cray MPI and Cray SHMEM

for Current and Next Generation Cray-XC Supercomputers. *Cray User Group (CUG)*, 2015, 2015.

[43] Douglas Doerfler, Brian Austin, Brandon Cook, Jack Deslippe, Krishna Kandalla, and Peter Mendygral. Evaluating the networking characteristics of the Cray XC-40 Intel Knights Landing-based Cori supercomputer at NERSC. *Concurrency and Computation: Practice and Experience*, 30(1):e4297, 2018.

[44] LibCRC. `https://www.libcrc.org/`.

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

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

[47] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.