A Parallel Multithreaded Sparse Triangular Linear System Solver

İlke Çuğu[a], Murat Manguoğlu[a,*]

[a]*Department of Computer Engineering, Middle East Technical University, 06800 Ankara, Turkey*

Abstract

We propose a parallel sparse triangular linear system solver based on the Spike algorithm. Sparse triangular systems are required to be solved in many applications. Often, they are a bottleneck due to their inherently sequential nature. Furthermore, typically many successive systems with the same coefficient matrix and with different right hand side vectors are required to be solved. The proposed solver decouples the problem at the cost of extra arithmetic operations as in the banded case. Compared to the banded case, there are extra savings due to the sparsity of the triangular coefficient matrix. We show the parallel performance of the proposed solver against the state-of-the-art parallel sparse triangular solver in Intel's Math Kernel Library (MKL) on a multicore architecture. We also show the effect of various sparse matrix reordering schemes. Numerical results show that the proposed solver outperforms MKL's solver in $\sim 80\%$ of cases by a factor of 2.47, on average.

Keywords: sparse triangular linear systems, direct methods, parallel computing
2010 MSC: 65F05, 65F50, 65Y05

1. Introduction

Many applications of science and engineering require the solution of large sparse linear systems. One well-known approach is to solve these systems by factorizing the coefficient matrix into nonsingular sparse triangular matrices and solving the resulting sparse triangular systems via backward and forward sweep (substitution) operations. This can be considered as a direct solver or if incomplete factorization is computed, it is part of the preconditioning in an iterative scheme. Common sparse factorizations that require the solution of sparse triangular systems include: LU, QR factorizations and their incomplete

*Corresponding author
Email address: `manguoglu@ceng.metu.edu.tr` (Murat Manguoğlu)
URL: `www.ceng.metu.edu.tr/~manguoglu` (Murat Manguoğlu)

counterparts (incomplete LU and incomplete QR). Furthermore, Gauss-Seidel and its variants such as Successive Over Relaxations (SOR) and Symmetric SOR, require the solution of a sparse triangular system at each iteration.

For large problems, solution of linear systems is often the most time consuming operations and in parallel computing platforms solution of triangular systems is known to scale worse than the factorization stage. They are often a sequential bottleneck due the dependencies between unknowns during forward and backward sweeps. Therefore, scalable parallel algorithms for solving sparse triangular linear systems are needed. Currently, there are many sparse triangular solver implementations available as standalone functions or within LU/ILU factorization software. The amount of interest in sparse triangular solvers is tremendous which is also seen by the number of available software packages. These include Euclid [1], Aztec [2], The Yale Sparse Matrix Package [3], SuperLU [4], HYPRE [5], PARDISO [6], PETSc [7], MUMPS [8], UMFPACK [9], PSBLAS [10], and PSPASES [11]. Along with the software packages, parallel triangular solvers are extensively studied in the literature for both MIMD (Multiple Instruction, Multiple Data) and SIMD (Single Instruction, Multiple Data) architectures. Most studies are focused on either level-scheduling [12, 13] or graph-coloring [14] algorithms. Level-scheduling algorithm is optimized for General Purpuse Graphical Processing Units (GPGPU) in [15, 16, 17], and for Central Processing Units (CPU) in [18, 19, 20, 21, 22]. Compared to level-scheduling, graph coloring is an NP-complete problem, therefore heuristics used for coloring may vary among the parallel solver implementations. Nevertheless, this idea is the basis for the solvers proposed in [23, 24] for GPGPUsmakes it also a considerable option for PSTRSV., and in [25, 26, 27] for CPUs. Apart from these two algorithms, in [28] a synchronization free algorithm on GPGPUs to overcome the barrier synchronization is presented by the level-scheduling and the analysis stage required to discover the underlying parallelism. In [29] a solver tailored for the sparsity structure arise in sparse Cholesky and LU factorizations is proposed, in which both dense and sparse solvers are utilized and assigned to different parts of a given triangular system. In [30] iterative solvers such as Jacobi and Block-Jacobi are proposed for solving sparse triangular systems with increased parallelism in exchange for a direct solution. On the other hand, several studies [31, 32, 19, 20, 33, 25, 34, 27] are focused on reordering the coefficient matrix beforehand to increase the available parallelism. In addition, [35, 31, 32, 36, 22] are focused on the effect of data layout on parallel sparse triangular system solver performance. Nevertheless, in all existing triangular solvers parallelism is limited by the inherited dependency in backward/forward sweep operations.

In this study, we propose a Spike [37] based parallel direct sparse triangular system solver. We implement the proposed algorithm for multicore shared address space architectures. The Spike algorithm is originally designed for banded linear systems [38, 39, 40, 41] and generalized for sparse linear systems first as a solver for banded preconditioner [42, 43] and later as the generalization of the banded spike algorithm for general sparse systems [44, 45, 46]. Furthermore, the banded Spike algorithm was implemented for GPU [47] and Multicore [48]

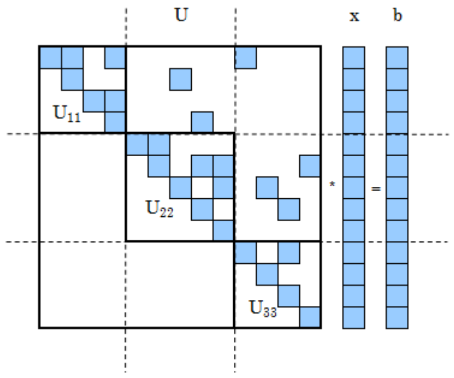

Figure 1: The sparse triangular linear system of $Ux = b$

architectures. Our work expands the algorithm for the sparse triangular case which differs significantly from the original banded triangular case. The concurrency available for the proposed solver is tightly coupled with the sparsity structure of the coefficient matrix. Hence, we also employ matrix reordering to improve parallelism, and use five well-known methods which are METIS [49, 50], Approximate Minimum Degree Permutation (AMD) [51], Column Permutation (ColPerm) in Matlab R2018a, Nested Dissection Permutation (NDP) [52, 53], and Reverse Cuthill-McKee Ordering (RCM) [53] in the experiments.

We describe the proposed parallel algorithm for the solution of sparse triangular linear systems in Section 2. Then, we analyze the performance constraints of the preprocessing and the solution stages in Section 3. Performance comparison of the proposed method and the parallel solver of Intel MKL is given in Section 4, and we conclude in Section 5.

2. Algorithm

The objective of the proposed algorithm is to solve sparse lower or upper triangular systems of equations in parallel. Without loss of generality assume a systems of equations is given,

$$Ux = b \tag{1}$$

where $U \in \mathbb{R}^{n\times n}$, full-rank, sparse upper triangular matrix. b and x are the right hand side and solution vectors, respectively.

The proposed parallel algorithm is based on the parallel Spike scheme in which the coefficient matrix is factorized into block diagonal matrix and the spike matrix. We refer the reader to the references in Section 1 for a more detailed description of the general and banded Spike factorizations.

In our case, the coefficient matrix is triangular and sparse. Hence, we have the following Spike factorization

$$U = DS \tag{2}$$

where D is block triangular with diagonal blocks that are also sparse and upper triangular, and S (illustrated in Figure 2) is upper triangular with identity main diagonal blocks and some dense columns (i.e. the spikes) in the upper off-diagonal blocks only. Given the linear system in Eq. 1 and the factorization in Eq. 2, the proposed algorithm can be described as follows. Assume that we multiply both sides of Eq. 1 with D^{-1} from left and obtain,

$$D^{-1}Ux = D^{-1}b. \tag{3}$$

Then, since

$$S = D^{-1}U, \tag{4}$$

we obtain the following modified system which has the same solution vector as the original system in Eq. 1,

$$Sx = g \tag{5}$$

where

$$g = D^{-1}b. \tag{6}$$

Note that obtaining the modified system is perfectly parallel in which there is no communication requirement. The key idea of the Spike algorithm is that the modified system contains a small reduced system (which does not exist in the original system in Eq. 1) that is independent from the rest of the unknowns. After solving this smaller reduced system, the solution of the original system can be also retrieved in perfect parallelism. The Spike algorithm was originally designed for the parallel computer architectures where the cost of arithmetic operations are much lower than the cost of interprocess communication and memory operations [39]. Today's multicore parallel architectures can perform arithmetic operations an order of magnitude faster, and this trend is not likely to change in the near future. Therefore, the arithmetic redundancy cost can be easily amortized and this observation is also valid for the sparse triangular case.

Now, we illustrate the proposed algorithm on a small (13×13) system given (without numerical values of nonzeros) in Figure 1. Given a partitioning of the coefficient matrix, we also partition the right hand side and the solution vectors, conformably. Next, we extract the block diagonal part of the coefficient matrix, such that,

$$U = D + R \tag{7}$$

where R contains the remaining nonzeros in the off-diagonal blocks. For the small example this is illustrated in Figure 3. In general, D is in the form of

$$D = \begin{pmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_t \end{pmatrix} \tag{8}$$

where t is the number of partitions (or threads) and each D_i is a separate independent $m_i \times m_i$ triangular matrix.

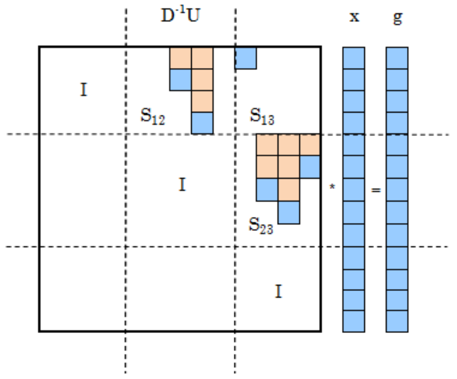

Figure 2: An example structure of the S matrix. The blue elements are from the original matrix where the orange ones represent the "spikes" resulted from $D^{-1}U$

The modified system in Eq. 5 contains a smaller independent reduced system,

$$\widehat{S}\widehat{x} = \widehat{g} \tag{9}$$

where $\widehat{x}$ corresponds to the dependencies in the original system (Figure 4).

We define i^{th} block row (R_i) as follows,

$$R_i = \left(0, .., 0, R_{i,i+1}, R_{i,i+2}, ..., R_{i,t}\right). \tag{10}$$

Furthermore, after identifying the bottom zero rows of R_i (if they exist), we define $\hat{R}_i$ as follows,

$$R_i = \begin{pmatrix} \hat{R}_i \\ 0 \end{pmatrix} \tag{11}$$

where the size of $\hat{R}_i$ is $k_i \times n$ with $k_i \leq m_i$. Note that k_i is determined by the sparsity structure of R_i. $\hat{R}_i$ determines the dependencies in partition i to other partitions if $k_i \neq 0$. Otherwise, the unknowns belonging to partition i are completely independent. Using Eq. 1 and 7 we obtain the following system,

$$Dx = b - Rx \tag{12}$$

where only those elements of x that are corresponding to nonzero columns of R are needed to compute the right hand side. We denote these elements of R in the nonzero columns as *dependency elements*. In fact, the reduced system in Eq. 9 can be formed by identifying the unknowns in x required by the *dependency elements*. Hence, for most cases both S and g only need to be computed partially (i.e. only $\widehat{S}$ and $\widehat{g}$ are needed). After solving the reduced system in Eq. 9, we update the right hand side of the system in Eq. 12 and solve it. Note that this last step involves solving independent triangular systems of equations since, unlike the original system, problem is decoupled now.

An important point is that after computing g in Eq. 6, some elements in x are already available without any further computations. This happens when

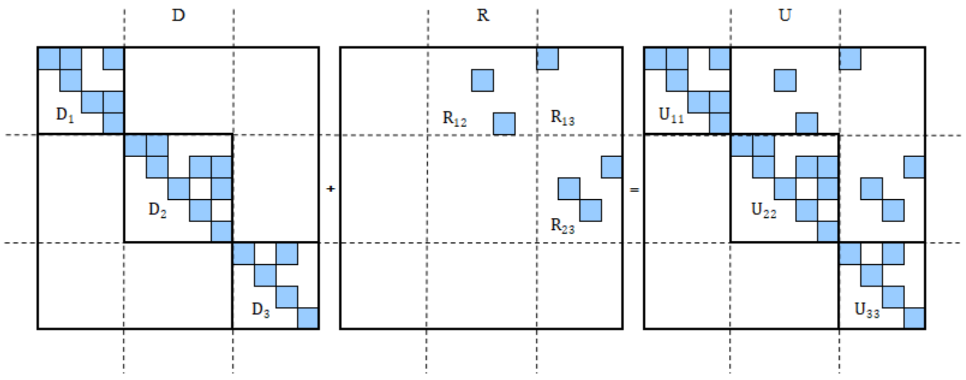

Figure 3: The illustration of $D + R = U$

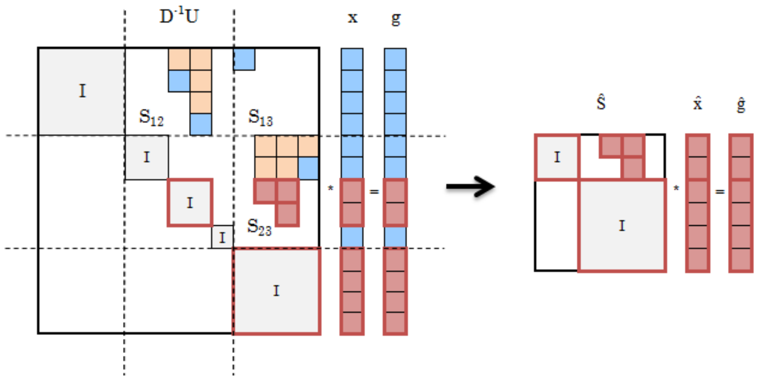

Figure 4: Construction of the reduced system

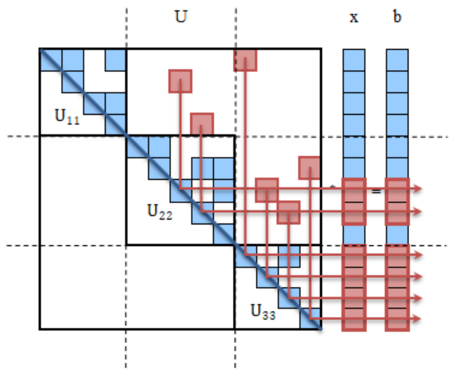

Figure 5: The illustration of light beams as dependency mappings.

$k_i < m_i$. If we split D_i matrix into two parts with respect to k_i, then the sub-matrix below the k_i will not have any corresponding *dependency elements*. In other words, let us denote the lower sub-matrix as $D_i^{(b)}$ from now on, the solution of

$$D_i^{(b)} g_i^{(b)} = b_i^{(b)} \tag{13}$$

directly gives the partial solution of the original system. Hence,

$$x_i^{(b)} = g_i^{(b)} \tag{14}$$

We further partition the upper part of g_i into two vectors with respect to a parameter we call "the reflection", r_i. If we think *dependency elements* as light sources sending light beams towards the bottom of the matrix and the diagonal as a mirror, then we can model the dependencies in a nonsingular triangular system as reflections of these light beams. These reflections are illustrated in Figure 5 and indicated as red arrows. The topmost arrow for each partition is selected as the reflection r_i and it shows the upper bound for the necessary part of each S_i matrix that we have to calculate to be able to form the reduced system $\widehat{S}$. Specifically, for

$$g_i = \begin{pmatrix} g_i^{(t)} \\ \hdashline g_i^{(m)} \\ \hdashline g_i^{(b)} \end{pmatrix} \begin{matrix} r_i - 1 \\ \hdashline k_i - r_i + 1 \\ \hdashline m_i - k_i \end{matrix} \tag{15}$$

where $r_i \leq k_i$, we do not need to make any calculations for $g_i^{(t)}$ vectors to construct $\widehat{S}$. In addition, if $r_i > k_i$, then $\widehat{x}_i = \widehat{g}_i$ since there is no "spike" within the range of row indices $[r_i, m_i]$. Our implementation takes $r_i = k_i$ when $r_i > k_i$ for simplification. If there is no reflection in the given partition, we set $hasReflection_i$ parameter as *false* and deem further partitioning of D_i (Eq. 16) as unnecessary.

Exploiting these properties saves us from recomputing $x_i^{(b)}$ and redundant operations with $g_i^{(t)}$. Therefore, we partition each D_i where *hasReflection*$_i$ is *true* as:

$$D_i = \begin{pmatrix} D_i^{(t)} & Q_i & P_i^{(t)} \\ & D_i^{(m)} & P_i^{(b)} \\ & & D_i^{(b)} \end{pmatrix}, \quad D_i^{(t;m)} = \begin{pmatrix} D_i^{(t)} & Q_i \\ & D_i^{(m)} \end{pmatrix}, \quad D_i^{(m;b)} = \begin{pmatrix} D_i^{(m)} & P_i^{(b)} \\ & D_i^{(b)} \end{pmatrix} \tag{16}$$

conformable with the partitioning of g_i vectors.

With these further partitions at hand, now, we can see that $\widehat{g}_i$ can be obtained via the solution of

$$D_i^{(m;b)} g_i^{(m;b)} = b_i^{(m;b)} \tag{17}$$

In detail, we select the elements of $g_i^{(m;b)}$, which are computed using the elements in $b_i^{(m;b)}$ that are hit by a light beam as in Figure 5, to form $\widehat{g}_i$. Then we solve the reduced system and update the corresponding elements in x.

$$\begin{aligned} \widehat{S}\widehat{x} &= \widehat{g} \\ x &\leftarrow \widehat{x} \end{aligned} \tag{18}$$

Then, we compute the new right-hand side vector for the independent triangular systems of $D_i^{(t;m)}$ partitions:

$$b_i^{(t;m)} := b_i^{(t;m)} - (\hat{R}_i x + P_i x_i^{(b)}) \tag{19}$$

where

$$P_i = \begin{pmatrix} P_i^{(t)} \\ P_i^{(b)} \end{pmatrix}. \tag{20}$$

The last step is to solve the isolated systems using the updated right-hand side without recomputing $x_i^{(b)}$:

$$D_i^{(t;m)} x_i^{(t;m)} = b_i^{(t;m)} \tag{21}$$

In order to achieve better load-balance, even if we do not have a reflection at a given partition (i.e. *hasReflection*$_i$ = *false*), we can still partition D_i with respect to k_i. Hence, we can solve Eq. 13 instead of waiting for idle while other threads are solving Eq. 17. However, we do this only if the performance drop in Eq. 17:

$$\begin{aligned} \lambda_{old}^{(1)} &= max\{nnz(D_i^{(m;b)}) | i \in \{1, ..., t\}, hasReflection_i\} \\ \lambda_{additional}^{(1)} &= max\{nnz(D_i^{(b)}) | i \in \{1, ..., t\}, \neg hasReflection_i\} \\ loss^{(1)} &= max(0, \lambda_{additional}^{(1)} - \lambda_{old}^{(1)}) \end{aligned} \tag{22}$$

is smaller than the overall gain in Eq. 19 and Eq. 21:

$$\begin{gathered}
\lambda^{(2)}_{old\_1} = max\{nnz(\hat{R}_i) + nnz(D_i) | i \in \{1, ..., t\}, \neg hasReflection_i\} \\
\lambda^{(2)}_{old\_2} = max\{nnz(\hat{R}_i) + nnz(P_i) + nnz(D_i^{(t;m)}) | i \in \{1, ..., t\}, hasReflection_i\} \\
\lambda^{(2)}_{old} = max(\lambda^{(2)}_{old\_1}, \lambda^{(2)}_{old\_2}) \\
\lambda^{(2)}_{new} = max\{nnz(\hat{R}_i) + nnz(P_i) + nnz(D_i^{(t;m)}) | i \in \{1, ..., t\}\} \\
gain^{(2)} = max(0, \lambda^{(2)}_{old} - \lambda^{(2)}_{new})
\end{gathered} \tag{23}$$

We add a small constant into the inequality and form the condition as:

$$gain^{(2)} > loss^{(1)} + \epsilon \tag{24}$$

If the condition in Eq. 24 is met, we proceed with the further partitioning of the D_i matrices for the threads with no reflection to improve the load-balance. In the implementation, we indicate this by setting *isOptimized*$_i$ parameter of a relevant thread as *true*. If R_i is an empty matrix, in other words $k_i = 0$, for thread i, then we select the best cut α_i preserving the condition in Eq. 24 and set $k_i = \alpha_i$. Note that we split the operations into the preprocessing and solution stages such that any operation that does not require the right hand side vector, b, constitutes the preprocessing stage. Remaining operations constitute the solution stage. This splitting is useful when multiple systems with the same coefficient matrix but different right hand side vectors are solved repeatedly, which is often the case in practice. The solution stage of PSTRSV is given in algorithm 1.

3. Performance constraints

In this section, we present key parameters that influence the performance of the proposed algorithm. These parameters are r_i, k_i, and the number of nonzeros in $\widehat{S}$. We analyze the performance for the preprocessing and solution stages separately.

3.1. Preprocessing

In preprocessing stage, we handle operations that are independent from the right hand side vector. This splitting is useful when it is used in an iterative scheme, preprocessing is done only once and the solver is often called multiple times. Hence, the cost of the preprocessing can usually be amortized. The operations involved in the preprocessing stage are the partitioning of D_i and R_i, computing S_i parts when necessary, building the reduced system, and investigation for a better load-balance. Among these, memory allocation and the computation required for S_i are the most significant performance bottleneck for the test matrices in the preprocessing time.

Algorithm 1 PSTRSV

```
Input: Partitioned and factored coefficient matrix $U = DS$, reduced coefficient matrix $\hat{S}$, together with associated dependency information and $b$, the right-hand side vector
Output: $x$, solution vector of $Ux = b$
for each thread $i = 1, 2, ..., t$ do
    if $hasReflection_i$ or $isOptimized_i$ then
        Solve the triangular system $D_i^{(m;b)} g_i^{(m;b)} = b_i^{(m;b)}$ for $g_i^{(m;b)}$
    end if
    Wait until all threads reach this point
    for a single thread $i$ do
        Solve the reduced system $\widehat{S}\widehat{x} = \widehat{g}$ for $\widehat{x}$
        Update the solution vector $x \leftarrow \widehat{x}$
    end for
    Wait until all threads reach this point
    if $hasDependence_i$ then
        $b_i^{(t;m)} := b_i^{(t;m)} - (\hat{R}_i x + P_i x_i^{(b)})$
    end if
    if $hasReflection_i$ or $isOptimized_i$ then
        Solve the triangular system $D_i^{(t;m)} x_i^{(t;m)} = b_i^{(t;m)}$ for $x_i^{(t;m)}$
    else
        Solve the triangular system $D_i x_i = b_i$ for $x_i$
    end if
end for
return $x$
```

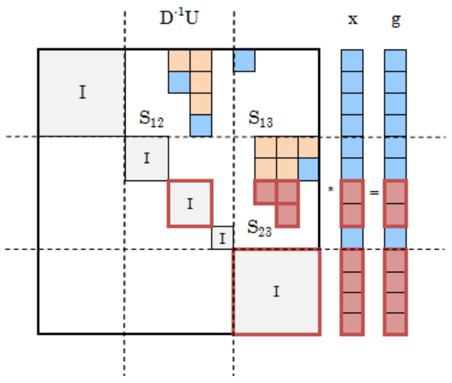

Figure 6: The dependencies which are present in the original system. We only need to compute the parts highlighted in red in order to construct the reduced system.

We only need the nonzeros of S_i within the range of row indices $[r_i, k_i]$ to build the reduced system (Figure 6). In Eq. 4, S has the following structure:

$$S_i = \left(0, ..., 0, I, S_{i,i+1}, S_{i,i+2}, ..., S_{i,t}\right). \tag{25}$$

If we ignore preceding zero blocks, we get

$$\hat{S}_i = \left(\begin{array}{ccc|c} I & & & \bar{S}_i^{(t)} \\ & I & & \bar{S}_i^{(b)} \\ & & I & 0 \end{array}\right) \tag{26}$$

conformable with the partitioning of g_i and R_i. In other words,

$$S_i = \left(0, \hat{S}_i\right) \tag{27}$$

Then, we can compute $\bar{S}_i$ by solving

$$D_i^{(t;m)} \bar{S}_i = \bar{R}_i \tag{28}$$

where

$$\bar{S}_i = \begin{pmatrix} \bar{S}_i^{(t)} \\ \bar{S}_i^{(b)} \end{pmatrix}, \quad \hat{R}_i = \left(0, \bar{R}_i\right) \tag{29}$$

Note that Eq. 28 is a triangular system with multiple right hand side vectors, $\bar{R}_i$. However, we do not need to compute $\bar{S}_i^{(t)}$ since it has no contribution to the reduced system. Therefore, we only solve a part of the system which is represented by the following equality,

$$D_i^{(m)} \bar{S}_i^{(b)} = \bar{R}_i^{(b)} \tag{30}$$

where

$$\bar{R}_i = \begin{pmatrix} \bar{R}_i^{(t)} \\ \bar{R}_i^{(b)} \end{pmatrix} \tag{31}$$

In the implementation, we transform $\bar{R}_i^{(b)}$ into a dense matrix containing only columns with at least one nonzero since $\bar{S}_i^{(b)}$ is expected to have dense spikes. We denote them as $\bar{R}_{dense_i}^{(b)}$ and $\bar{S}_{dense_i}^{(b)}$ respectively. Let d_i be the number of columns in R_i having at least one nonzero. Then, $\bar{S}_{dense_i}^{(b)}$ is a $(k_i - r_i + 1) \times d_i$ dense matrix which is computed only if $r_i \leq k_i$. In other words, for a matrix where $r_i > k_i, \forall i \in \{1, 2, ..., t\}$ there is no memory allocation or computational cost for $\bar{R}_{dense_i}^{(b)}$ and $\bar{S}_{dense_i}^{(b)}$ matrices. Naturally, this also holds if $d_i = 0, \forall i \in \{1, 2, ..., t\}$ since having no *dependency element* is the ideal scenario for parallelism. Nevertheless, it is still beneficial to have a relatively small value of $max\{k_i - r_i | i \in \{1, 2, ..., t\}\}$ for $d_i \neq 0$ considering the dense structure of the spikes. In addition to the mathematical dynamics, for maximum performance in practice, instead of creating large parallel tasks by distributing $\bar{S}^{(b)}_{dense_i}$ computations (Eq. 30) to t threads, we split each $\bar{S}^{(b)}_{dense_i}$ to t column-based partitions and leverage parallelism at this level. In other words, we sequentially iterate through $\bar{S}^{(b)}_{dense_i}$ computations and split each one to create t smaller parallel tasks to improve the overall preprocessing performance. This approach combines coarse grained partitioning with fine grained processing, hence it is suitable for multilevel cache hierarchies of modern multicore architectures.

3.2. Solution

In the solution stage, we have two parallel regions and a sequential region (Eq. 18) between them. We can optimize the performance of these two parallel regions using the load-balance strategy explained in Section 2. This leaves us with Eq. 18 where we solve the reduced system and update the solution vector.

The coefficient matrix $\widehat{S}$ of the reduced system is a $d \times d$ unit diagonal sparse triangular matrix where d is at most the sum of all d_i explained in Section 3.1:

$$d \leq \sum_{i=1}^{t} d_i \tag{32}$$

since d_i values through partitions may contain duplicated columns. Solving the reduced system takes $\mathcal{O}(nnz(\widehat{S}) - d)$ operations. Again, for $d_i = 0, \forall i \in \{1, 2, ..., t\}$ there is no reduced system, so we have perfect parallelism. However, for most cases where $d \neq 0$, the sparsity structure of U determines the number of off-diagonal nonzeros in $\widehat{S}$. For a matrix where $r_i > k_i, \forall i \in \{1, 2, ..., t\}$, $\widehat{S}$ is the identity matrix. Hence, there is no need to solve the reduced system,

$$\begin{gathered} \widehat{S} = I, \text{ when } r_i > k_i, \forall i \in \{1, 2, ..., t\} \\ I\widehat{x} = \widehat{g} \text{ from Eq. 9} \\ \widehat{x} = \widehat{g} \end{gathered} \tag{33}$$

and if we directly store g_i vectors in x_i parts before forming $\widehat{g}$, then there is no memory operation for updating the solution vector either. If $r_i \leq k_i, \exists i \in \{1, 2, ..., t\}$, then the computational cost will be determined by the sparsity structure of the *dependency elements* within the range of row indices $[r_i, k_i]$.

4. Numerical results

We perform numerical experiments to demonstrate the parallel scalability of the proposed algorithm against the multithreaded double precision sparse triangular system solver (mkl_sparse_d_trsv) of Intel MKL 2018 [54]. Hereafter, we refer to them as PSTRSV and MKL, respectively. We have obtained twenty real-world test matrices from the SuiteSparse Matrix Collection [55] that arise in variety of application areas and have a variety of dimensions/nonzeros (see Table 1 for properties and the application domains that they arise in).

#	Matrix	Dimension(n)	Non-zeros(nnz)	Application
1.	Dubcova2	65, 025	1, 030, 225	2D/3D Problem
2.	Dubcova3	146, 689	3, 636, 643	2D/3D Problem
3.	FEM_3D_thermal1	17, 880	430, 740	Thermal Problem
4.	G3_circuit	1, 585, 478	7, 660, 826	Circuit Simulation
5.	apache2	715, 176	4, 817, 870	Structural Sim.
6.	bmwcra_1	148, 770	10, 641, 602	Structural Problem
7.	boneS01	127, 224	5, 516, 602	Model Reduction
8.	c-70	68, 924	658, 986	Optimization
9.	c-big	345, 241	2, 340, 859	Optimization
10.	consph	83, 334	6, 010, 480	2D/3D Problem
11.	ct20stif	52, 329	2, 600, 295	Structural Problem
12.	ecology2	999, 999	4, 995, 991	2D/3D Problem
13.	engine	143, 571	4, 706, 073	Structural Problem
14.	filter3D	106, 437	2, 707, 179	Model Reduction
15.	finan512	74, 752	596, 992	Economic Problem
16.	parabolic_fem	525, 825	3, 674, 625	Fluid Dynamics
17.	pwtk	217, 918	11, 524, 432	Structural Problem
18.	shallow_water1	81, 920	327, 680	Fluid Dynamics
19.	torso3	259, 156	4, 429, 042	2D/3D Problem
20.	venkat50	62, 424	1, 717, 777	Fluid Dynamics

Table 1: Properties of the test matrices.

As we have mentioned in Section 3, the sparsity structure of the triangular matrix is expected to have a significant influence on the performance of triangular solvers. Therefore, for both PSTRSV and MKL, we experiment with five well-known matrix reordering schemes. These are METIS [49, 50], Approximate Minimum Degree Permutation (AMD) [51], Column Permutation (ColPerm of Matlab R2018a), Nested Dissection Permutation (NDP) [52, 53], and Reverse Cuthill-McKee Ordering (RCM) [53]. After applying the permutation, we remove the strictly lower triangular part of the matrix to obtain U matrix. For reorderings that require symmetric matrices, when we have an unsymmetric test matrix A, we apply the reordering to the matrix $A^T + A$, then the resulting permutation is used on the original matrix, A. For all test problems, we use a random right hand side vector.

We use a computer with 2 sockets and 2 Intel(R) Xeon(R) CPU E5-2650 v3 processors each having 10 cores and 16 GB of memory. Threads are distributed using "KMP_AFFINITY = granularity = fine,compact,1,0". Matrices are stored in Compressed Sparse Row (CSR) format and the proposed solver is implemented using C programming language with OpenMP [56]. We repeat each run 1, 000 times and obtain the average of the required wallclock time. Preprocessing time excludes reordering time since it is common for both algorithms.

Algorithm 2 STRSV

```
**Input:** $U$ matrix in CSR format and $b$, the right-hand side vector
**Output:** $x$, solution vector of $Ux = b$
$x[n-1] = b[n-1]/u[iu[n-1]]$
**for** $i = n-2, n-3, ..., 0$ **do**
    $t = b[i]$
    **for** $j = iu[i]+1, iu[i]+2, ..., iu[i+1]-1$ **do**
        $t := t - u[j] * x[ju[j]]$
    **end for**
    $x[i] = t/u[iu[i]]$
**end for**
**return** $x$
```

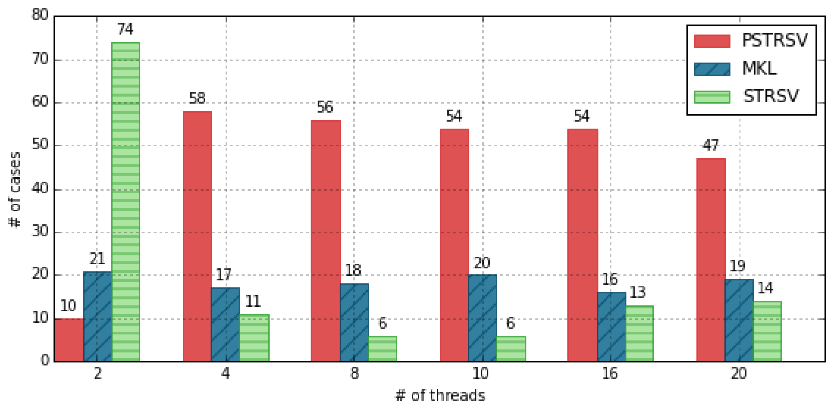

Figure 7: Overall performance comparison of the proposed solver, Intel MKL and the best sequential solver. Bars indicate the number of test cases where the given solver outperforms others. We ignore the test cases where we are unable to evaluate the performance due to memory constraints.

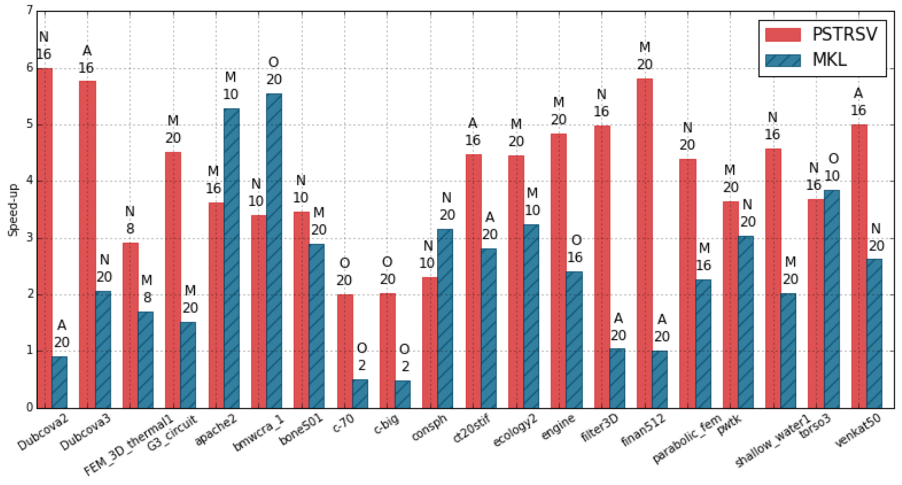

Figure 8: The highest speed-ups achieved by the proposed solver and Intel MKL solver. {R: RCM, C: ColPerm, N: NDP, M: METIS, A: AMD, O: ORIGINAL} symbols on bars indicate the matrix reordering algorithms which give the best result. The thread counts are placed under them.

For performance overview, we present the number of test cases where the fastest solution is provided by a particular triangular solver in Figure 7. In a number of cases, we were not able to run solvers for a particular test matrix or its reordered version due to memory constraints. Hence, we have only 6 cases where we are able to measure the performance for all of the reorderings we mentioned along with the original matrix using each thread count $t \in \{2, 4, 8, 10, 16, 20\}$. For these 6 test cases, we present the speed-up curves in Figures 10, 11, 12, 13, 14, and 15. However, we give the best speed-up achieved by PSTRSV and MKL for all matrices in Figure 8. In this chart, we show only the best speed-up achieved for a given test matrix as well as the matrix reordering and number of threads being used to achieve the best speedup. The final residuals obtained by PSTRSV are comparable with MKL.

The speedup (s) is computed against the baseline sequential time. The baseline is either our custom sequential sparse triangular solver implementation (algorithm 2) or the sequential solver in Intel MKL whichever is the fastest for the given problem;

$$s = \frac{min(runtime_{custom}, runtime_{MKL})}{runtime_{parallel}} \tag{34}$$

In general, PSTRSV provides the best speedup for most of the test cases. This can be observed in Figure 7 where PSTRSV is better than others in 65% of the test cases on average for $t > 2$. Furthermore, in Figure 8, we present the highest performance improvements achieved for each of the 20 test matrices. PSTRSV outperforms MKL in 80% of the test cases and is 2.3 times faster on average.

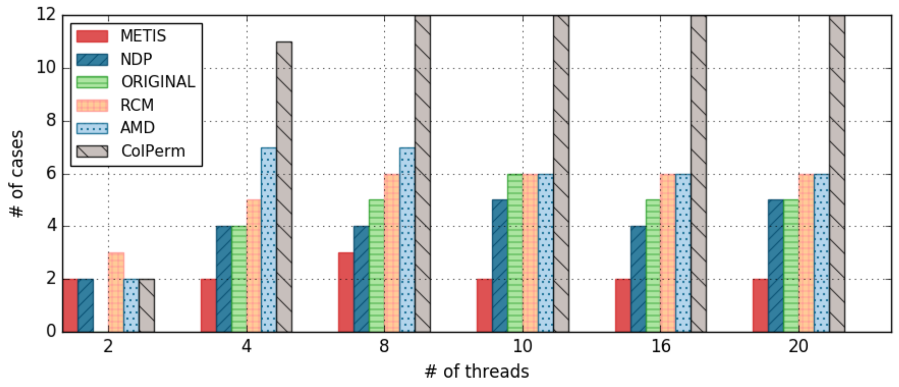

Figure 9: The number of cases where the employed reordering algorithms get memory error.

Based on the results, PSTRSV benefits most from the parallelism provided by NDP in 9/20 cases, METIS in 6/20 cases, and AMD in 3/20 cases. For the other 2 cases, the original coefficient matrix gave the best results. ColPerm and RCM, on the other hand, are not suitable for both PSTRSV and MKL. We note that for all 20 test matrices, there is at least one reordering that does not fail as shown in Figure 8. Furthermore, in Figure 9, we depict the cases and reordering methods for which the memory was not enough as the number of partitions is increased. In our implementation, we use the same number of threads as the number of partitions. As expected, the results show that METIS partitioning is less prone to high memory consumption as it is a heuristic that directly attempts to minimize the number of off-diagonal entries in the partitioned matrix while others do not have such objective. In addition, although NDP may cause higher memory consumption than METIS for some problems, considering the possibility of achieving a superior speedup with NDP makes it also a favorable option for PSTRSV.

Now, we look into those 6 cases where all reordering schemes work in more detail. In Figure 10 (*ct20stif*), using NDP, METIS and AMD, PSTRSV outperforms MKL by obtaining a speedup of $\sim 4\times$. Using RCM, ColPerm, and ORIGINAL reorderings, MKL performs slightly better than PSTRSV, while the speedup is poor (< 2). In Figure 11 (*FEM_3D_thermal1*), for all methods the speedup is poor. PSTRSV outperforms MKL only in NDP case by reaching $\sim 2.5\times$ speed-up. In Figure 12 (*finan512*), PSTRSV outperforms MKL in all cases except ColPerm, where both perform poorly. The best speedup attained by PSTRSV is ~ 6. MKL consistently produces < 1 speedup for all cases. In Figure 13 (*pwtk*), with NDP, METIS, and AMD, PSTRSV outperforms MKL by reaching a speedup of ~ 3. Poor parallelism with RCM results in worse performance than MKL which is able to reach $\sim 2\times$ speed-up. In Figure 14 (*shallow_water1*), PSTRSV achieves a good speedup regardless the reordering

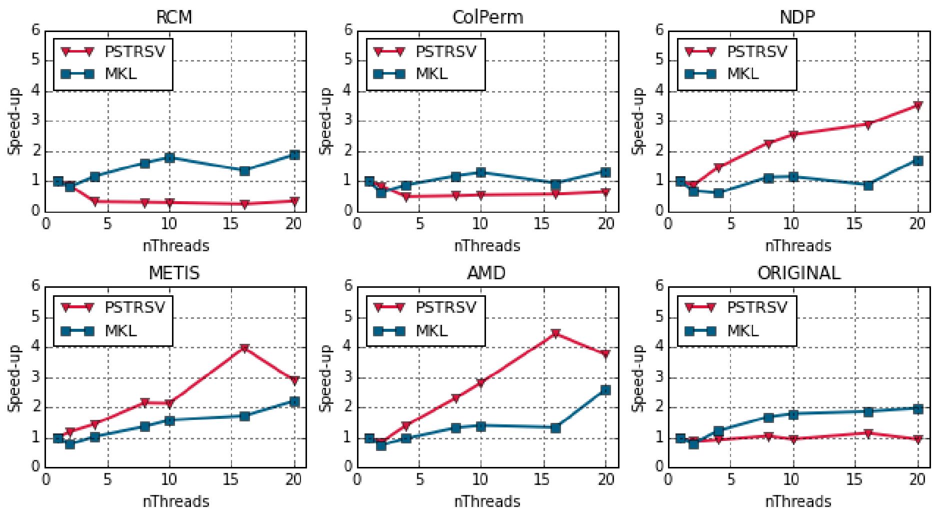

Figure 10: The speed-up comparison for ct20stif

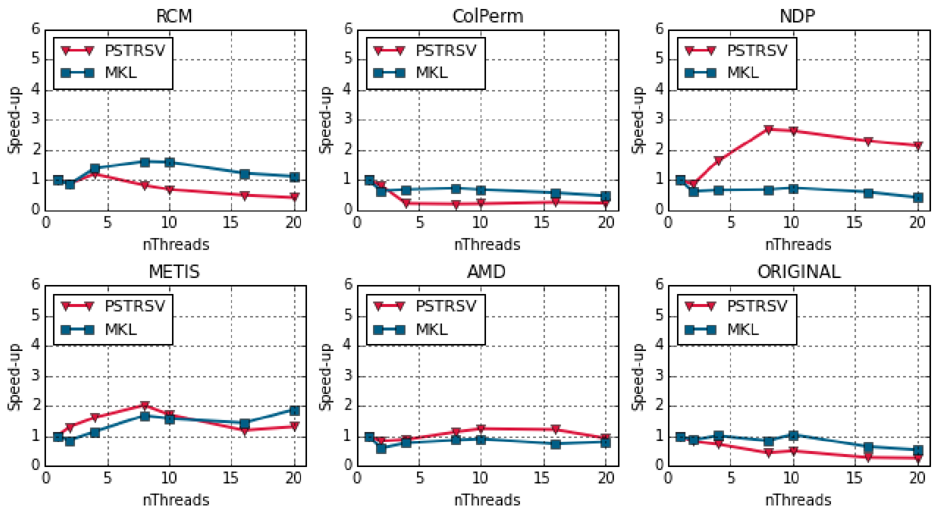

Figure 11: The speed-up comparison for FEM_3D_thermal1

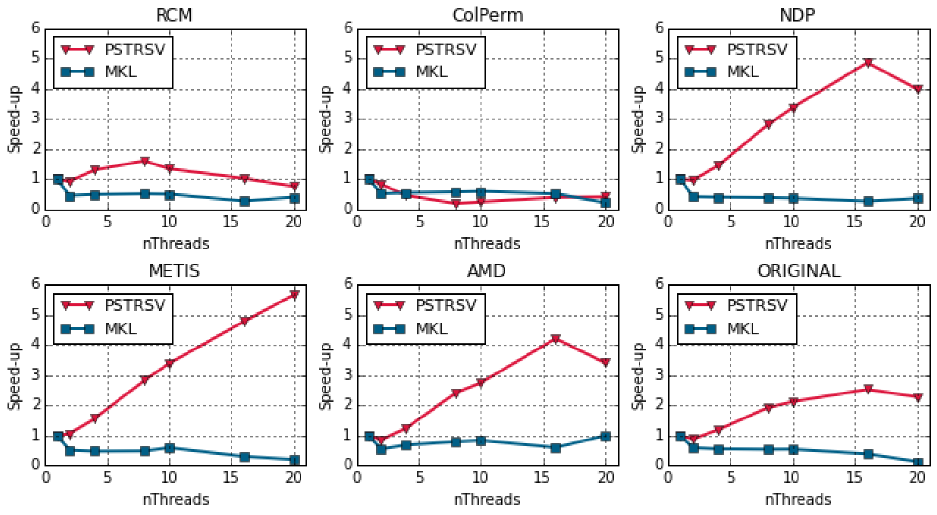

Figure 12: The speed-up comparison for finan512

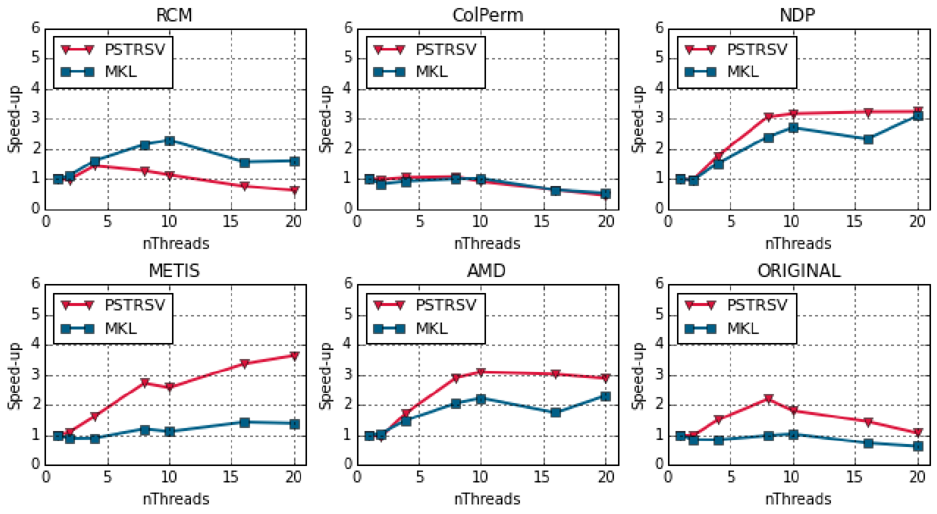

Figure 13: The speed-up comparison for pwtk

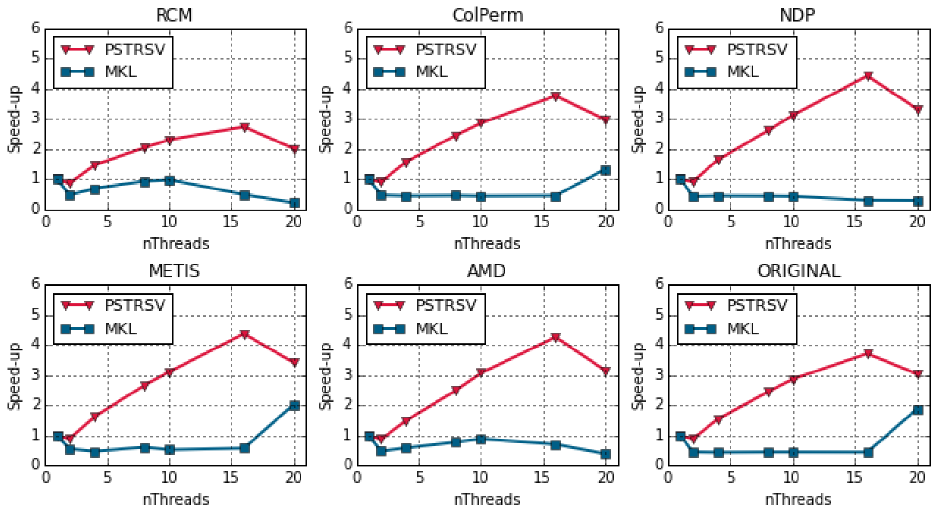

Figure 14: The speed-up comparison for shallow_water1

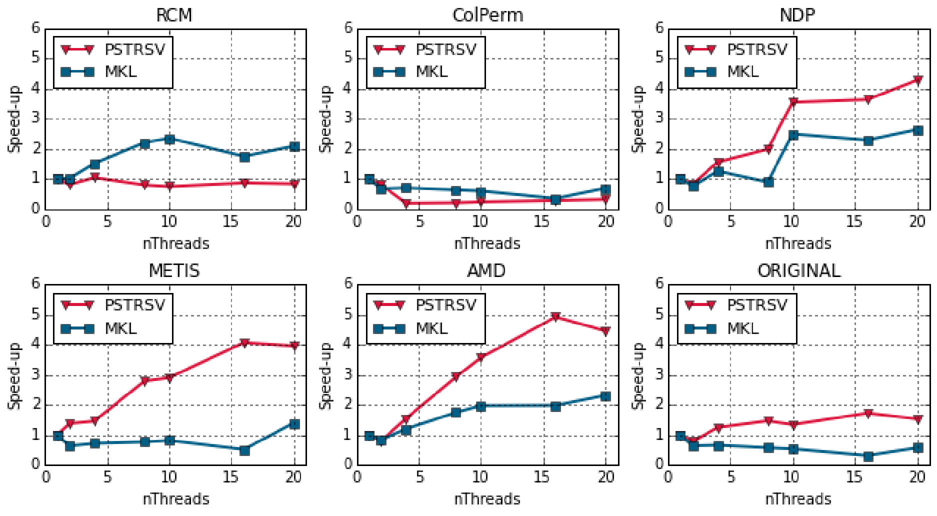

Figure 15: The speed-up comparison for venkat50

t	PSTRSV				MKL			
	min	max	avg	std	min	max	avg	std
2	1.19	37.83	13.52	9.65	4.11	251.50	78.77	60.23
4	2.28	2111.62	319.87	402.90	2.82	131.36	46.50	37.07
8	2.83	1167.28	227.06	256.19	2.17	114.80	32.89	27.84
10	2.99	824.10	197.22	210.96	2.58	118.37	31.32	27.63
16	3.03	762.25	192.87	201.35	0.19	115.57	27.41	22.40
20	3.07	770.03	188.94	199.06	0.44	264.46	35.85	35.65

Table 2: Statistics of the preprocessing times of PSTRSV and MKL in milliseconds where t is the number of threads.

method. PSTRSV outperforms MKL in all cases by a factor of ~ 4. For *venkat50* (Figure 15), using NDP, METIS, AMD, and ORIGINAL, PSTRSV outperforms MKL by reaching at most $\sim 5\times$ speed-up. Again, poor parallelism with RCM results in a worse performance than MKL which is able to reach $\sim 2\times$ speedup.

So far, we have only looked into the solution time which excludes the preprocessing time. Now, we study the required number of iterations to amortize the preprocessing time. First, we give some statistics of preprocessing times required by both PSTRSV and MKL in Table 2. Note that preprocessing stage of PSTRSV is parallel which is reflected as a decrease in the average preprocessing times in Table 2 as increasing the number of threads (t). When $t = 2$, $r_0 = 0$ and $k_1 = 0$ which results in a relatively low preprocessing time since there is no cost regarding $\bar{R}^{(b)}_{dense_i}$ and $\bar{S}^{(b)}_{dense_i}$ matrices as explained in Section 3.1. The relatively high standard deviation in preprocessing times of PSTRSV indicates that PSTRSV is more sensitive to sparsity structure than MKL. Even though the cost of preprocessing for PSTRSV is relatively high, it can be amortized by the fast triangular solution stage. In Table 3, we give the number of iterations required by the proposed algorithm to amortize the preprocessing time against the best sequential solver. Note that, we only compute the required number of iterations only for those cases where PSTRSV has a speed-up $s > 1$ since, otherwise, it would require infinite amount of iterations. The parallelism available in preprocessing stage also affects amortization positively. Consistent with the Table 2, average iteration count required for amortization drops as number of threads are increased (for $t > 2$). In addition, we include the statistics of the required iteration counts for the cases presented in Figure 8, where we show the highest speed-ups achieved by the optimal settings. According to these results, PSTRSV is able to amortize the preprocessing times in approximately 40 iterations on average when the best strategy is selected regarding the number of threads and the reordering algorithm. Although, overall, MKL requires less preprocessing time than PSTRSV, it cannot amortize the preprocessing time in 21/120 test cases for any $t \in \{2, 4, 8, 10, 16, 20\}$, whereas PSTRSV cannot

t	min	max	avg	std
2	14	160	52.39	47.45
4	11	2326	292.46	394.37
8	10	1392	155.01	250.47
10	9	1589	98.29	207.90
16	9	429	85.98	86.75
20	9	1108	82.38	74.97
Cases in Fig 8	9	140	39.67	36.80

Table 3: Required iteration counts by PSTRSV for amortization of the preprocessing time where t is the number of threads.

amortize the preprocessing time only in 9/120 test cases.

5. Conclusions

In this paper, we presented a Spike based parallel sparse triangular linear system solver. We defined the key performance parameters of the proposed algorithm and analyzed their effect in terms of solution time. As test problems, we used matrices obtained from the SuiteSparse Matrix Collection that arise in real world applications and applied five well-known matrix reordering schemes. Experimental results show that the proposed algorithm benefits from METIS, AMD and NDP reorderings. The proposed algorithm outperforms parallel sparse triangular solver of Intel MKL 2018 on a multicore arhitecture.

References

[1] D. Hysom, A. Pothen, A scalable parallel algorithm for incomplete factor preconditioning, SIAM Journal on Scientific Computing 22 (6) (2001) 2194–2215.

[2] S. Hutchinson, J. Shadid, R. Tuminaro, Aztec user's guide. version 1, Tech. rep. (oct 1995). `doi:10.2172/135550`.
URL `https://doi.org/10.2172/135550`

[3] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, A. H. Sherman, Yale sparse matrix package i: The symmetric codes, International Journal for Numerical Methods in Engineering 18 (8) (1982) 1145–1151. `doi:10.1002/nme.1620180804`.
URL `https://doi.org/10.1002/nme.1620180804`

[4] X. S. Li, J. W. Demmel, Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Transactions on Mathematical Software (TOMS) 29 (2) (2003) 110–140.

[5] R. D. Falgout, U. M. Yang, hypre: A library of high performance preconditioners, in: International Conference on Computational Science, Springer, 2002, pp. 632–641.

[6] O. Schenk, K. Gärtner, W. Fichtner, A. Stricker, Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation, Future Generation Computer Systems 18 (1) (2001) 69–78.

[7] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik, et al., Petsc users manual revision 3.8, Tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States) (2017).

[8] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, J. Koster, A fully asynchronous multifrontal solver using distributed dynamic scheduling, SIAM Journal on Matrix Analysis and Applications 23 (1) (2001) 15–41.

[9] T. A. Davis, I. S. Duff, An unsymmetric-pattern multifrontal method for sparse lu factorization, SIAM Journal on Matrix Analysis and Applications 18 (1) (1997) 140–158.

[10] S. Filippone, M. Colajanni, Psblas: A library for parallel linear algebra computation on sparse matrices, ACM Transactions on Mathematical Software (TOMS) 26 (4) (2000) 527–550.

[11] M. Joshi, G. Karypis, V. Kumar, A. Gupta, F. Gustavson, Pspases: An efficient and scalable parallel sparse direct solver, in: In Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, Citeseer, 1999.

[12] E. Anderson, Y. Saad, Solving sparse triangular linear systems on parallel computers, International Journal of High Speed Computing 1 (01) (1989) 73–95.

[13] J. H. Saltz, Aggregation methods for solving sparse triangular systems on multiprocessors, SIAM Journal on Scientific and Statistical Computing 11 (1) (1990) 123–144.

[14] R. Schreiber, W.-P. Tang, Vectorizing the conjugate gradient method, Proceedings of the Symposium on CYBER 205 Applications.

[15] M. Naumov, Parallel incomplete-lu and cholesky factorization in the preconditioned iterative methods on the gpu, Tech. rep., NVIDIA Corp., Westford, MA, USA (2012).

[16] A. Picciau, G. E. Inggs, J. Wickerson, E. C. Kerrigan, G. A. Constantinides, Balancing locality and concurrency: solving sparse triangular systems on gpus, in: 2016 IEEE 23rd International Conference on High-Performance Computing (HiPC), IEEE, 2016, pp. 183–192.

[17] R. Li, Y. Saad, Gpu-accelerated preconditioned iterative linear solvers, The Journal of Supercomputing 63 (2) (2013) 443–466.

[18] J. Park, M. Smelyanskiy, N. Sundaram, P. Dubey, Sparsifying synchronization for high-performance shared-memory sparse triangular solver, in: International Supercomputing Conference, Springer, 2014, pp. 124–140.

[19] M. M. Wolf, M. A. Heroux, E. G. Boman, Factors impacting performance of multithreaded sparse triangular solve, in: International Conference on High Performance Computing for Computational Science, Springer, 2010, pp. 32–44.

[20] E. Rothberg, A. Gupta, Parallel iccg on a hierarchical memory multiprocessor addressing the triangular solve bottleneck., Parallel Computing 18 (7) (1992) 719 – 741.

[21] S. W. Hammond, R. Schreiber, Efficient iccg on a shared memory multiprocessor, International Journal of High Speed Computing 4 (01) (1992) 1–21.

[22] X. Wang, W. Xue, W. Liu, L. Wu, swsptrsv: a fast sparse triangular solve with sparse level tile layout on sunway architectures, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, 2018, pp. 338–353.

[23] M. Naumov, P. Castonguay, J. Cohen, Parallel graph coloring with applications to the incomplete-lu factorization on the gpu, Tech. rep., NVIDIA Corp., Westford, MA, USA (2015).

[24] B. Suchoski, C. Severn, M. Shantharam, P. Raghavan, Adapting sparse triangular solution to gpus, in: 2012 41st International Conference on Parallel Processing Workshops, IEEE, 2012, pp. 140–148.

[25] T. Iwashita, H. Nakashima, Y. Takahashi, Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in iccg method, in: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, IEEE, 2012, pp. 474–483.

[26] S. Ma, Y. Saad, Distributed ilu(0) and sor preconditioners for unstructured sparse linear systems, Tech. rep., Army High Performance Computing Research Center (1994).

[27] D. P. Koester, S. Ranka, G. C. Fox, A parallel gauss-seidel algorithm for sparse power system matrices, in: Proceedings of the 1994 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, 1994, pp. 184–193.

[28] W. Liu, A. Li, J. Hogg, I. S. Duff, B. Vinter, A synchronization-free algorithm for parallel sparse triangular solves, in: European Conference on Parallel Processing, Springer, 2016, pp. 617–630.

[29] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, K. A. Yelick, Automatic performance tuning and analysis of sparse triangular solve, in: In ICS 2002: Workshop on Performance Optimization via High-Level Languages and Libraries, 2002.

[30] E. Chow, H. Anzt, J. Scott, J. Dongarra, Using jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning, Journal of Parallel and Distributed Computing 119 (2018) 219230. doi:10.1016/j.jpdc.2018.04.017.
URL http://www.sciencedirect.com/science/article/pii/S0743731518303034

[31] E. Totoni, M. T. Heath, L. V. Kale, Structure-adaptive parallel solution of sparse triangular linear systems, Parallel Computing 40 (9) (2014) 454–470.

[32] J. Mayer, Parallel algorithms for solving linear systems with sparse triangular matrices, Computing 86 (4) (2009) 291.

[33] X. S. Li, Evaluation of sparse lu factorization and triangular solution on multicore platforms, in: International Conference on High Performance Computing for Computational Science, Springer, 2008, pp. 287–300.

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

[35] B. Smith, H. Zhang, Sparse triangular solves for ilu revisited: data layout crucial to better performance, The International Journal of High Performance Computing Applications 25 (4) (2011) 386–391.

[36] K. Teranishi, P. Raghavan, E. Ng, A new data-mapping scheme for latency-tolerant distributed sparse triangular solution, in: Supercomputing, ACM/IEEE 2002 Conference, IEEE, 2002, pp. 27–27.

[37] A. H. Sameh, R. P. Brent, Solving triangular systems on a parallel computer, SIAM Journal on Numerical Analysis 14 (6) (1977) 1101–1113.

[38] S.-C. Chen, D. J. Kuck, A. H. Sameh, Practical parallel band triangular system solvers, ACM Transactions on Mathematical Software (TOMS) 4 (3) (1978) 270–277.

[39] J. J. Dongarra, A. H. Sameh, On some parallel banded system solvers, Parallel Computing 1 (3-4) (1984) 223–235.

[40] E. Polizzi, A. H. Sameh, A parallel hybrid banded system solver: the spike algorithm, Parallel computing 32 (2) (2006) 177–194.

[41] E. Polizzi, A. Sameh, Spike: A parallel environment for solving banded linear systems, Computers & Fluids 36 (1) (2007) 113–120.

[42] M. Manguoglu, A. H. Sameh, O. Schenk, Pspike: A parallel hybrid sparse linear system solver, in: European Conference on Parallel Processing, Springer, 2009, pp. 797–808.

[43] O. Schenk, M. Manguoglu, A. Sameh, M. Christen, M. Sathe, Parallel scalable pde-constrained optimization: antenna identification in hyperthermia cancer treatment planning, Computer Science-Research and Development 23 (3-4) (2009) 177–183.

[44] M. Manguoglu, A domain-decomposing parallel sparse linear system solver, Journal of Computational and Applied Mathematics 236 (3) (2011) 319–325.

[45] M. Manguoglu, Parallel solution of sparse linear systems, in: High-Performance Scientific Computing, Springer, 2012, pp. 171–184.

[46] E. S. Bolukbasi, M. Manguoglu, A multithreaded recursive and nonrecursive parallel sparse direct solver, in: Advances in Computational Fluid-Structure Interaction and Flow Simulation, Springer, 2016, pp. 283–292.

[47] I. E. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, A. H. Sameh, A direct tridiagonal solver based on givens rotations for gpu architectures, Parallel Computing 49 (2015) 101–116.

[48] K. Mendiratta, E. Polizzi, A threaded spike algorithm for solving general banded systems, Parallel Computing 37 (12) (2011) 733–741.

[49] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1) (1998) 359–392.

[50] G. Karypis, V. Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, Journal of Parallel and Distributed Computing 48 (1) (1998) 71–95.

[51] P. R. Amestoy, T. A. Davis, I. S. Duff, An approximate minimum degree ordering algorithm, SIAM Journal on Matrix Analysis and Applications 17 (4) (1996) 886–905.

[52] A. George, Nested dissection of a regular finite element mesh, SIAM Journal on Numerical Analysis 10 (2) (1973) 345–363.

[53] A. George, J. W. Liu, Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall Professional Technical Reference, 1981.

[54] Intel math kernel library. reference manual, Tech. rep., Intel Corporation, Santa Clara, USA (2018).
URL `https://software.intel.com/en-us/mkl`

[55] T. A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Transactions on Mathematical Software (TOMS) 38 (1) (2011) 1.

[56] L. Dagum, R. Menon, Openmp: an industry standard api for shared-memory programming, IEEE Computational Science and Engineering 5 (1) (1998) 46–55.