PARALLEL SOLUTION OF SPARSE TRIANGULAR LINEAR SYSTEMS ON MULTICORE PLATFORMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

İLKE ÇUĞU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

NOVEMBER 2018

Approval of the thesis:

PARALLEL SOLUTION OF SPARSE TRIANGULAR LINEAR SYSTEMS ON MULTICORE PLATFORMS

submitted by **İLKE ÇUĞU** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** ________________

Assoc. Prof. Dr. Murat Manguoğlu
Supervisor, **Computer Engineering Department, METU** ________________

Examining Committee Members:

Prof. Dr. Cevdet Aykanat
Computer Engineering Department, Bilkent University ________________

Assoc. Prof. Dr. Murat Manguoğlu
Computer Engineering Department, METU ________________

Assist. Prof. Dr. Hande Alemdar
Computer Engineering Department, METU ________________

Date: ________________

ABSTRACT

PARALLEL SOLUTION OF SPARSE TRIANGULAR LINEAR SYSTEMS ON MULTICORE PLATFORMS

Çuğu, İlke
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Murat Manguoğlu

November 2018, 93 pages

Many large-scale applications in science and engineering require the solution of 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 it is part of the preconditioning operation in an iterative scheme if incomplete factorization is computed. Often, these sparse triangular systems are the main performance bottleneck due to their inherently sequential nature. With the emergence of multi-core platforms, the interest in solving sparse triangular linear systems effectively in parallel has grown. In this thesis, a parallel sparse triangular linear system solver based on the generalization of Spike algorithm is proposed. The performance constraints of the proposed algorithm and their impacts on the performance are evaluated on matrices from different application domains. Furthermore, performance comparisons are made against the state-of-the-art parallel sparse triangular solver of Intel's Math Kernel Library.

Keywords: Sparse Triangular Linear Systems, Direct Solution, Parallel Computing

ÖZ

ÇOK ÇEKİRDEKLİ MİMARİLERDE SEYREK ÜÇGEN DOĞRUSAL SİSTEMLERİN PARALEL ÇÖZÜMÜ

Çuğu, İlke
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Murat Manguoğlu

Kasım 2018 , 93 sayfa

Bilim ve mühendislikteki pek çok uygulama seyrek doğrusal sistemlerin çözümüne ihtiyaç duyar. Doğrusal sistemleri çözmenin en iyi bilinen yöntemlerinden biri onları üçgensel çarpanlarına ayırıp bu üçgensel sistemleri çözmektir. Üçgensel doğrusal sistemler gerek doğrudan yöntemlere gerekse yinelemeli önkoşullamalara çözüm sağlar ya da tekrar tekrar işlenerek verilen problemleri çözüme yaklaştırırlar. Seri çözümlere uygun doğaları nedeniyle bu seyrek üçgensel doğrusal sistemlerin çözümü genelde paralel çözümlerdeki verimin ana belirleyicisidir. Çok çekirdekli mimarilerin yaygınlaşmasıyla seyrek üçgensel doğrusal sistemleri paralel olarak çözme eğilimi artmıştır. Bu tez çalışmasında, seyrek üçgensel doğrusal sistemlerin, Spike algoritmasına dayalı paralel çözümü tanıtılmıştır. Algoritmanın performans karakteristikleri ve bunların etkileri çeşitli uygulama alanlarından matrisler kullanılarak test edilmiştir. Ek olarak, İntel'in Temel Matematik Kütüphanesinde bulunan paralel seyrek üçgensel doğrusal sistem çözücü ile karşılaştırmalar yapılmıştır.

Anahtar Kelimeler: Seyrek Üçgensel Doğrusal Sistemler, Doğrudan Çözüm, Paralel İşlem

To my family

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis advisor, Assoc. Prof. Dr. Murat Manguoğlu for his guidance and encouragement. I have done two summer internships and this thesis work with him where he gave me the discipline of rigorous experimentation, and provided the freedom I need to work productively. He is very delicate and serious about his work, so his approval for this thesis is highly valuable.

During my graduate studies, I have also worked on machine learning with Asst. Prof. Dr. Emre Akbaş. I would like to thank him for taking interest in a student's research ideas and transforming them into research projects. His open-minded and progressive nature created a fruitful research environment for me.

Special thanks goes to my ultimate research associate, Eren Şener. I worked with him in every machine learning project I got involved. He endured every failure and annoyance in those projects, and he was always ready to start working on the next one.

I want to thank my thesis committee members Prof. Dr. Cevdet Aykanat and Asst. Prof. Dr. Hande Alemdar for the feedback they provided.

I would like to thank Dr. Christian Blug and Dr. Itır Önal Ertuğrul for providing references during my PhD applications. In addition, I thank Maja Pavlovic for her surprising support for my PhD applications.

Before proceeding any futher, I thank the quickest wit I know, Mr. Ender Gör.

I also want to thank Zülal Öztürk, Alişan Tosun, Furkan Onursal, Çağrı Erciyes, and Sinan Sarıoğlu because why not.

Finally, I would like to thank my father Fahrettin Çuğu, my mother Eda Danişe Çuğu, my elder sister Gülay Çuğu Bal, and my aunt Hülya Bilal for providing a peaceful and supportive environment throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

MIMD	Multiple Instruction, Multiple Data
SIMD	Single Instruction, Multiple Data
SOR	Successive Over-Relaxation
GPGPU	General Purpose Graphics Processing Unit
CPU	Central Processing Unit
AMD	Approximate Minimum Degree
NDP	Nested Dissection Permutation
RCM	Reverse Cuthill-McKee
CM	Cuthill-McKee
ColPerm	Column Permutation
BFS	Breath-First Search
MKL	Math Kernel Library
STRSV	Sparse Triangular System Solver
PSTRSV	Parallel Sparse Triangular System Solver
CSR	Compressed Sparse Row
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
ICCG	Incomplete Cholesky Conjugate Gradient
ILU	Incomplete LU Factorization
BLAS	Basic Linear Algebra Subprograms
CUDA	Compute Unified Device Architecture

CHAPTER 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 could also be considered as a 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 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, not only solution of linear systems is often the most time consuming operation, but also in parallel computing platforms solution of triangular systems is less scalable compared to the factorization. Solution of triangular systems 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 softwares. 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]. Furthermore, sparse matrix operations started to appear also in widely used machine learning frameworks such as Tensorflow [12], Caffe2 [13], PyTorch [14], Theano [15], and MXNet [16].

Along with the software packages, parallel triangular solvers are extensively studied in the literature for both MIMD and SIMD architectures. Most parallel solutions are focused on level-scheduling [17, 18] and graph-coloring [19] algorithms with a few exceptions where algorithms are tailored for the specific conditions arise in targeted problem domains. These algorithms address the mathematical nature of dependencies between elements and try to solve the given sparse triangular system efficiently, but they do not attempt to change the sparsity structure that prevents effective parallelism. Hence, they are often combined with matrix reordering algorithms to increase the available parallelism. In this setting, given coefficient matrices are first reordered by a matrix ordering algorithm, then the reordered system is solved by a parallel sparse triangular solver. In addition, one of the challenges in parallel sparse matrix operations is the poor memory coalescing caused by the sparse matrix rows with varying number of nonzeros. Therefore, effective data layouts for sparse triangular systems are also investigated in the literature.

In this thesis, we propose a Spike [20] 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 [21, 22, 23, 24] and generalized for sparse linear systems first as a solver for banded preconditioner [25, 26] and later as the generalization of the banded spike algorithm for general sparse systems [27, 28, 29]. Furthermore, the banded Spike algorithm was implemented for GPGPU [30] and Multicore [31] architectures. Our work expands and specialize 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 [32, 33], Approximate Minimum Degree Permutation (AMD) [34], Column Permutation (ColPerm) in Matlab R2018a, Nested Dissection Permutation (NDP) [35, 36], and Reverse Cuthill-McKee Ordering (RCM) [36] in the numerical experiments.

We first summarize the parallel sparse triangular solver literature and explain the employed matrix reordering algorithms in Chapter 2. We describe the proposed parallel algorithm for the solution of sparse triangular linear systems in Chapter 3. Then, we analyze the performance constraints of the preprocessing and the solution phases in Chapter 4. Performance comparison of the proposed method and the parallel solver of Intel MKL is given in Chapter 5, and we conclude in Chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, first, we give some background information about the matrix ordering algorithms employed in this thesis to explore the effect of different reordering approaches on the parallel performance. Second, we summarize and categorize the parallel sparse triangular solver algorithms found in the literature.

2.1 Matrix Ordering

The concurrency available for parallel sparse triangular system solvers are tightly coupled with the sparsity structure of the coefficient matrix of a given linear system. Hence, in the literature, several studies [37, 38, 39, 40, 41, 42, 43, 44] are focused on reordering the coefficient matrix beforehand to increase the available parallelism. In this thesis, we also employ matrix reordering to improve parallelism, and use METIS [32, 33], AMD [34], ColPerm, NDP [35, 36] and RCM [36] during the experiments. Note that for METIS, AMD and RCM which require symmetric matrices, we apply the reordering to the matrix $(|A|^T + |A|)$ when we have an unsymmetric test matrix A, then the resulting permutation is used on the original matrix A to produce the reordered version. In this section, we briefly discuss the employed matrix ordering algorithms.

Nested Dissection Permutation algorithm is proposed by Alan George [35] in 1973, and re-factored by Alan George and Joseph W. Liu [36] in 1981. NDP is a graph separator algorithm in which the coefficient matrix is transformed into a graph and it is split into subgraphs that are not connected. In other words, the algorithm recursively finds a separator and cuts the given graph into two halves with nearly equal sizes.

This reordering is particularly useful for the proposed algorithm since it pushes the *dependency elements* (which are explained in Chapter 3) towards the boundaries of the partitions and maximizes the *reflection* r_i (see Chapter 3) parameters. We explain the benefit of having large r_i values in Chapter 4, and present the empirical evidence in Chapter 5.

Reverse Cuthill-McKee algorithm is proposed by Alan George and Joseph W. Liu [36] in 1981. It is a simple improvement over the original algorithm, which is designed by Elizabeth Cuthill and James McKee [45] in 1969, to reduce the fill-in even further. RCM is a variant of the standard breath-first search algorithm. It introduces a strict traversal policy to the BFS algorithm in which adjacent nodes are visited in ascending vertex order. During the traversal, each visited node is inserted into the result set R. At the end, R indicates the new order of the vertices. In RCM this result set is reversed, and that is the only difference between RCM and CM.

METIS [32] is a software package developed in Karypis Lab, which contains serial or parallel (ParMETIS [33]) programs for graph partitioning and fill-reducing sparse matrix ordering. We used the multilevel k-way partitioning scheme in METIS Version 5.1.0 during our experiments. Specifically, we selected the communication volume minimization mode in which METIS tries to gather the nonzeros near the main diagonal of the coefficient matrix.

Approximate Minimum Degree ordering algorithm is proposed by Patrick R. Amestoy, Timothy A. Davis and Iain S. Duff [34] in 1996 which is an extension over the original minimum degree algorithm proposed by William F. Tinney and John W. Walker [46] in 1967. Unlike the original one, AMD does not compute the exact vertex degrees instead it computes an upper bound to approximately set the degrees of the vertices. The algorithm is one of the most widely used fill-reducing heuristics. Briefly, the coefficient matrix is again taken as a graph and AMD iterates through the given graph in a greedy fashion where the next node with the smallest approximate degree is selected and eliminated in each step.

ColPerm is a sparse column permutation algorithm available in Matlab2018a. It produces a permutation vector to order the columns of the given coefficient matrix according to increasing number of nonzeros.

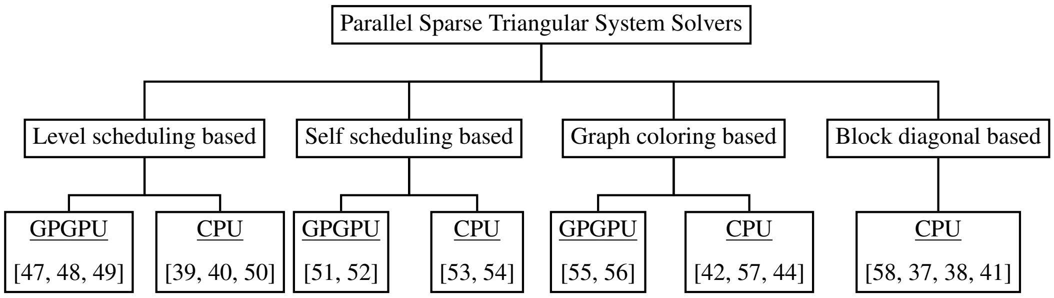

Figure 2.1: Taxonomy of parallel direct sparse triangular system solvers

2.2 Parallel Sparse Triangular Solvers

In this section, we categorize and briefly discuss the parallel sparse triangular system solvers found in the literature for both MIMD and SIMD architectures. Generic performance improvements on sparse triangular system solvers such as the data layout optimization in [59] are not covered since they do not specifically propose parallel algorithms. In addition, since the proposed algorithm offers a direct solution to the given triangular linear system, we will focus on the direct solvers rather than the iterative solvers such as [60, 61] where Jacobi and Block-Jacobi iterations are proposed for solving sparse triangular systems with increased parallelism in exchange for a direct solution.

We give the taxonomy tree of the direct solvers in Figure 2.1. In this tree, we group the studies in the literature under four main categories. These groups are defined as level-scheduling [17, 18], self-scheduling [62], graph-coloring [19], and block diagonal based methods. In these categories, level-scheduling and self-scheduling methods are rooted from the same idea of treating the coefficient matrix as a directed acyclic graph and representing the dependencies as levels, but they differentiate on whether barrier synchronization is employed or not. Compared to others, graph-coloring methods are focused on the reordering of the coefficient matrix to exploit concurrency which is not explicitly available in the original form of the triangular system. For the algorithms that are not using any level construction or coloring, we found a common idea of processing block diagonals as isolated systems and treating the rest as dependencies between these systems. Therefore, we labeled them as block diagonal based methods and finalized the taxonomy tree. Note that, in some studies, combination of different categories in a hybrid parallel solver is proposed or conjectured as more effective than plain approaches.

2.2.1 Level-scheduling Based Methods

Level scheduling algorithm is first introduced by Anderson and Saad [17], and later by Saltz [18]. It forms levels of rows by exploring the dependencies in the coefficient matrix by treating it as a directed acyclic graph. Concurrency is achieved within levels by processing the rows in parallel. However, the levels are processed sequentially. This algorithm consists of two phases, called analysis and solve. In analysis phase, levels are formed by traversing the graph representation of the coefficient matrix, and in the solve phase, the sparse triangular system is solved by using the level representation. In general usage, the solve phase is called multiple times in an iterative solver after a single analysis phase.

Naturally, earlier studies implemented level-scheduling algorithm for CPUs. In [40], the sparse triangular solve is deemed as the main bottleneck for the ICCG algorithm. Therefore, a level-scheduling based parallel algorithm is proposed for the triangular solution which is accompanied with a matrix reordering phase for the coefficient matrix to solve the performance problem caused by the poor spatial locality of the data. Moreover, in [39], level-scheduling algorithm is tested for different thread affinities and barrier types. In the implementation of the analysis phase, they used a variant of BFS to form the levels, and, as an improvement over the original algorithm, they permuted the system symmetrically with respect to the levels to sort the rows/columns in order of the levels. For the solution phase, they propose the usage of barriers that use spin-locks and active polling to improve the performance. The most recent work on level-scheudling [50] introduced a new data layout, named Sparse Level Tile layout, to improve the data reuse of the right hand side and solution vectors. It is stated that the proposed layout may introduce more levels to a given problem. However, the performance drop caused by the extra levels are solved by utilizing fast register communication for level synchronization.

Recently, level-scheduling algorithm is also adapted to GPGPUs. The first implementation of this kind is proposed in [47] and its BFS based analysis phase is integrated into parallel ILU and Cholesky factorizations in [63]. Another study [49] improved the parallel performance of level-scheduling algorithm by replacing the row-levels with subgraph levels to increase the data locality. In addition, a new matrix storage format named HEC (Hybrid ELL and CSR) [64] is adapted for the solution phase of the level-scheduling algorithm in [48] to increase the effective bandwidth of the GPGPU.

2.2.2 Self-scheduling Based Methods

Self-scheduling [62] is a modification over the level-scheduling scheme where the barrier synchronization between levels are replaced with individual waiting mechanisms. In other words, each processing unit waits for its direct dependency to be computed and immediately starts to work upon receiving the result or notification even if the others in the same level are still waiting.

As in level-scheduling case, earlier studies implemented this approach for CPUs. In [54], a self-scheduling scheme for the triangular solution part of the ICCG is proposed with a dynamic work sharing between processors. Another CPU implementation is proposed in [53] where they run three operations after the construction of the levels to improve the parallel performance. First, they eliminate the dependency edges between the elements that are assigned to the same thread since they will naturally execute in program order. Second, they combine tasks into supertasks to reduce the number of dependency edges. Third, they remove the transitive edges since they are already covered by the execution flow.

Some of the recent work on parallel sparse triangular system solvers managed to deploy the self-scheduling idea to GPGPUs. In [52] a synchronization free algorithm based on spin-locks is proposed to overcome the barrier synchronization in the level-scheduling. In addition, their method requires a simple preprocessing phase where they only compute the in-degree of each vertex. On the other hand, another study [51] directly focused on the level-scheduling implementation in CUDA, and proposed a self-scheduling based alternative in which the modified BFS is replaced with a parallel topological sorting algorithm to set the levels, and the barrier synchronization is replaced with a counter-based scheduling mechanism where each element only waits for its own dependencies.

2.2.3 Graph-coloring Based Methods

Graph coloring algorithm [19] tries to assign the minimum number of colors to vertices of a graph in a way that two neighboring vertices are not allowed to have the same color. Compared to others, graph coloring is an NP-complete problem, therefore heuristics that are used for coloring may vary among the parallel solver implementations. Over the years, several studies explored the possible implementation of graph-coloring to increase the parallelism of sparse triangular solvers.

In [57], authors used graph multi-coloring for the effective distribution of computational workload between processors. In which, the coefficient matrix is reordered according to the computed row colors. They used this graph partitioning scheme in the parallel triangular solve phases of ILU(0), Block SOR and Symmetric SOR. In a contemporary study [44], a multi-coloring algorithm based on the saturation degree ordering algorithm [65] is proposed to improve the performance of parallel Gauss-Seidel iterations. This algorithm is specifically designed for the last diagonal matrix blocks resulted from the block-diagonal-bordered ordering [66] applied on power system matrices. In a recent CPU implementation [42], authors proposed algebraic block multicolor ordering which is an improvement over the block multicolor ordering [67] for the coefficient matrix of the triangular solve in the ICCG method. In this scheme, resulting matrix blocks with the same color are solved in parallel and each thread process one or more of these blocks, but the computation within a block is sequential.

Graph-coloring methods are also investigated for SIMD architectures. In [56], the level-scheduling based approach in [47] is outperformed by a graph-coloring based parallel sparse triangular solver implementation in CUDA. They devised a coloring scheme in which each colored group of rows depends only one or more previous groups. Moreover, it is hypothesized that combining this graph-coloring approach with level-scheduling may improve the overall performance. The idea of developing a hybrid approach is proved to be useful in [55] where graph coloring based on finding the maximum independent set [68] is combined with level-scheduling for ILU factorization.

2.2.4 Block-diagonal Based Methods

In this section, we propose a new category, called block diagonal based methods, for the parallel sparse triangular solvers in the literature. The governing dynamics for these solvers are the isolated triangular systems in the form of block diagonals within a coefficient matrix. In general, these isolated systems are solved by sequential sparse triangular solvers simultaneously. The perfect parallelism is prevented by the off-diagonal parts. Hence, parallel solutions in the literature are mostly focused on effective messaging structures, matrix partitioning procedures, and workload sharing policies. This seems particularly suitable for CPUs since we did not found a GPGPU counterpart that can be considered as a block diagonal based method.

In the literature, we have found several studies that can be named under this category. For example, in [58] a parallel sparse triangular 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. Moreover, the parallel sparse triangular solver in SuperLU_DIST [41] also employed the block diagonal approach. Specifically, during the solution, when a dependent element is computed the owner processor send the result to the ones that are waiting for it. After receiving the dependent element, each processor computes the local sum, and at the end the diagonal processor performs the division. In another study [38], two algorithms, called *block anti diagonal* and *anti diagonal column* algorithms, are proposed. In these algorithms, the coefficient matrix is partitioned into diagonal blocks and rectangular off-diagonal blocks. The diagonal blocks are processed sequentially whereas the rectangular blocks are processed in parallel. Finally, a structure adaptive algorithm [37] is proposed. This algorithm identifies the independent rows in the coefficient matrix and groups them together via reordering. Then, it analyzes the structure of the reordered matrix and distributes the workload accordingly. Provided they exist, it processes the dense off-diagonal blocks by using highly tuned dense BLAS operations in separate processes. In addition to this algorithm, they built an prioritized messaging scheme between processes to send the computed dependent elements right away while handling diagonal blocks. As a side note, since the computation in sparse triangular solve is very small relative to the amount of data, they deemed cache inefficiencies as intolerable and processed the rows in large chunks.

The proposed algorithm in Chapter 3 can be considered as a block diagonal based method.

CHAPTER 3

THE PROPOSED 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,

$$Ux = b \tag{3.1}$$

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 designed 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 Chapter 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

$$U = DS \tag{3.2}$$

where D is block triangular with diagonal blocks that are also sparse and upper triangular, and S (illustrated in Figure 3.2) 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. 3.1 and the factorization in Eq. 3.2, the proposed algorithm can be described as follows. Assume, we multiply both sides of Eq. 3.1 with D^{-1} from left and obtain,

$$D^{-1}Ux = D^{-1}b. \tag{3.3}$$

Then, since

$$S = D^{-1}U, \tag{3.4}$$

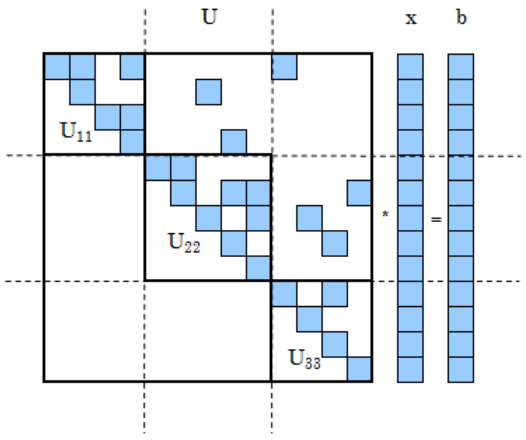

Figure 3.1: The sparse triangular linear system of $Ux = b$

we obtain the following modified system which has the same solution vector as the original system in Eq. 3.1,

$$Sx = g \tag{3.5}$$

where

$$g = D^{-1}b. \tag{3.6}$$

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. 3.1) 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 [22]. 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 3.1. 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,

$$U = D + R \tag{3.7}$$

where R is the remaining nonzeros in the off-diagonal blocks. For the small example this is illustrated in Figure 3.3. In general, D is in the form of

$$D = \begin{pmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_t \end{pmatrix} \tag{3.8}$$

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

The modified system in Eq. 3.5 contains a smaller independent reduced system,

$$\widehat{S}\widehat{x} = \widehat{g} \tag{3.9}$$

where $\widehat{x}$ corresponds to the dependencies in the original system (Figure 3.4).

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

$$R_i = \Big(0, .., 0, R_{i,i+1}, R_{i,i+2}, ..., R_{i,t}\Big). \tag{3.10}$$

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

$$R_i = \begin{pmatrix} \hat{R}_i \\ 0 \end{pmatrix} \tag{3.11}$$

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. 3.1 and 3.7 we obtain the following system,

$$Dx = b - Rx \tag{3.12}$$

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. 3.9 can be formed

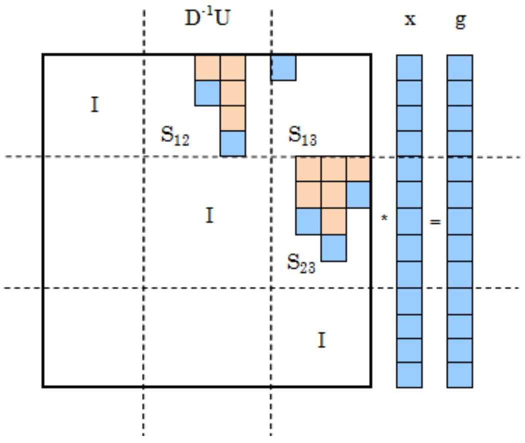

Figure 3.2: An example structure of the S matrix. The blue elements are from the original matrix where the orange ones represent the "spikes" resulted from $D^{-1}U$

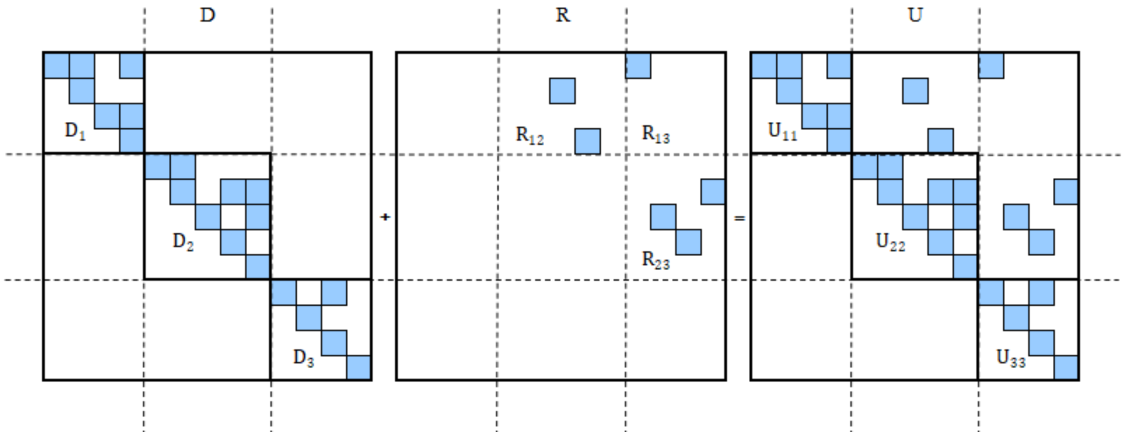

Figure 3.3: The illustration of $D + R = U$

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. 3.9, we update the right hand side of the system in Eq. 3.12 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. 3.6, some elements in x are already available without any further computations. This happens when $k_i < m_i$. In order to elaborate, 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,

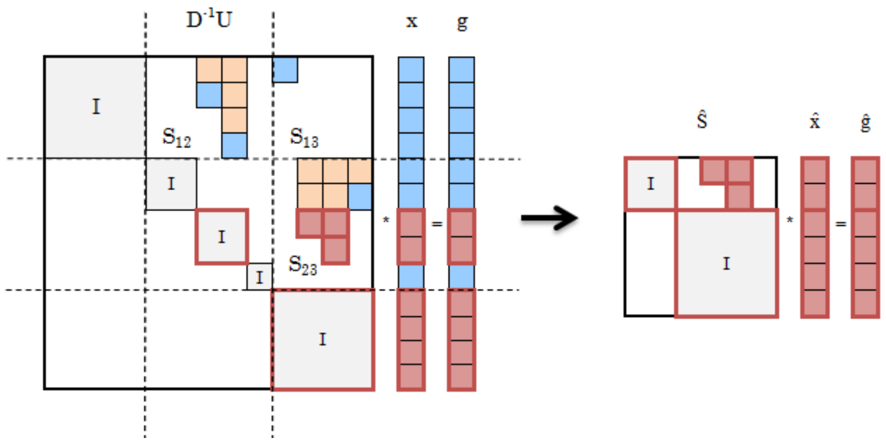

Figure 3.4: Construction of the reduced system

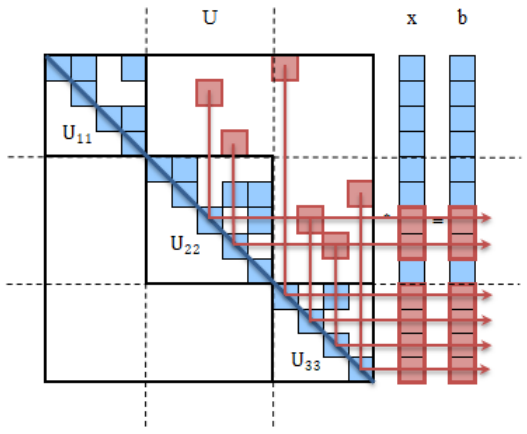

Figure 3.5: The illustration of light beams as dependency mappings.

let us denote the lower sub-matrix as $D_i^{(b)}$ from now on, the solution of

$$D_i^{(b)} g_i^{(b)} = b_i^{(b)} \tag{3.13}$$

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

$$x_i^{(b)} = g_i^{(b)} \tag{3.14}$$

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 3.5 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

$$g_i = \begin{pmatrix} g_i^{(t)} \\ \hdashline g_i^{(m)} \\ \hdashline g_i^{(b)} \end{pmatrix} \begin{array}{c} r_i \\ \hdashline k_i - r_i \\ \hdashline m_i - k_i \end{array} \tag{3.15}$$

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. 3.16) 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:

$$D_i = \begin{pmatrix} D_i^{(t)} & Q_i & P_i^{(t)} \\ & D_i^{(m)} & P_i^{(b)} \\ & & D_i^{(b)} \end{pmatrix}, \quad D_i^{(t;m)} = \begin{pmatrix} D_i^{(t)} & Q_i \\ & D_i^{(m)} \end{pmatrix}, \quad D_i^{(m;b)} = \begin{pmatrix} D_i^{(m)} & P_i^{(b)} \\ & D_i^{(b)} \end{pmatrix} \tag{3.16}$$

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

$$D_i^{(m;b)} g_i^{(m;b)} = b_i^{(m;b)} \tag{3.17}$$

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 3.5, to form $\widehat{g}_i$. Then we solve the reduced system and update the corresponding elements in x.

$$\begin{gathered} \widehat{S}\widehat{x} = \widehat{g} \\ x \leftarrow \widehat{x} \end{gathered} \tag{3.18}$$

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

$$b_i^{(t;m)} := b_i^{(t;m)} - (\hat{R}_i x + P_i x_i^{(b)}) \tag{3.19}$$

where $P_i = \begin{pmatrix} P_i^{(t)} \\ P_i^{(b)} \end{pmatrix}$

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

$$D_i^{(t;m)} x_i^{(t;m)} = b_i^{(t;m)} \tag{3.20}$$

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. 3.13 instead of waiting for idle while other threads are solving Eq. 3.17. However, we do this only if the performance drop in Eq. 3.17:

$$\begin{gathered}
\lambda_{old}^{(1)} = max\{nnz(D_i^{(m;b)}) | i \in \{1, ..., t\}, \textit{hasReflection}_i\} \\
\lambda_{additional}^{(1)} = max\{nnz(D_i^{(b)}) | i \in \{1, ..., t\}, \neg\textit{hasReflection}_i\} \\
loss^{(1)} = max(0, \lambda_{additional}^{(1)} - \lambda_{old}^{(1)})
\end{gathered} \tag{3.21}$$

is smaller than the overall gain in Eq. 3.19 and Eq. 3.20:

$$\begin{gathered}
\lambda_{old\_1}^{(2)} = max\{nnz(\hat{R}_i) + nnz(D_i) | i \in \{1, ..., t\}, \neg\textit{hasReflection}_i\} \\
\lambda_{old\_2}^{(2)} = max\{nnz(\hat{R}_i) + nnz(P_i) + nnz(D_i^{(t;m)}) | i \in \{1, ..., t\}, \textit{hasReflection}_i\} \\
\lambda_{old}^{(2)} = max(\lambda_{old\_1}^{(2)}, \lambda_{old\_2}^{(2)}) \\
\lambda_{new}^{(2)} = max\{nnz(\hat{R}_i) + nnz(P_i) + nnz(D_i^{(t;m)}) | i \in \{1, ..., t\}\} \\
gain^{(2)} = max(0, \lambda_{old}^{(2)} - \lambda_{new}^{(2)})
\end{gathered} \tag{3.22}$$

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

$$gain^{(2)} > loss^{(1)} + \epsilon \tag{3.23}$$

If the condition in Eq. 3.23 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. 3.23 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.

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 $x$
```

CHAPTER 4

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.

4.1 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 for 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.

We only need the nonzeros of S_i within the range of row indices $[r_i, k_i]$ to build the reduced system (Figure 4.1). In Eq. 3.4, S has the following structure:

$$S_i = \Big(0, ..., 0, I, S_{i,i+1}, S_{i,i+2}, ..., S_{i,t}\Big)\,. \tag{4.1}$$

If we ignore preceding zero blocks, we get

$$\hat{S}_i = \left(\begin{array}{ccc|c} I & & & \bar{S}_i^{(t)} \\ & I & & \bar{S}_i^{(b)} \\ & & I & 0 \end{array}\right) \tag{4.2}$$

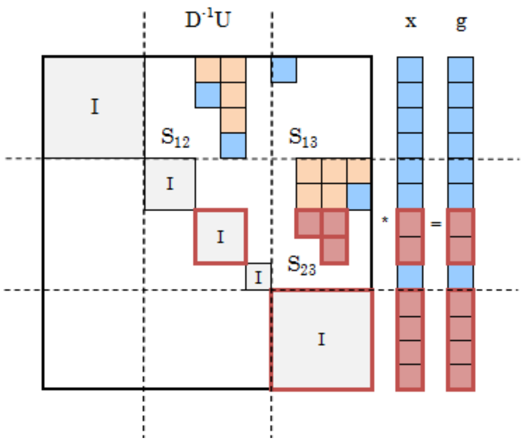

Figure 4.1: The dependencies presented in the original system. We only need to calculate S matrix parts highlighted in red to construct the reduced system.

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

$$S_i = \left(0, \hat{S}_i\right) \tag{4.3}$$

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

$$D_i^{(t;m)} \bar{S}_i = \bar{R}_i \tag{4.4}$$

where

$$\bar{S}_i = \begin{pmatrix} \bar{S}_i^{(t)} \\ \bar{S}_i^{(b)} \end{pmatrix}, \quad \hat{R}_i = \left(0, \bar{R}_i\right) \tag{4.5}$$

Note that Eq. 4.4 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,

$$D_i^{(m)} \bar{S}_i^{(b)} = \bar{R}_i^{(b)} \tag{4.6}$$

where

$$\bar{R}_i = \begin{pmatrix} \bar{R}_i^{(t)} \\ \bar{R}_i^{(b)} \end{pmatrix} \tag{4.7}$$

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, $\bar{S}^{(b)}_{dense_i}$ 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 $\bar{R}^{(b)}_{dense_i}$ and $\bar{S}^{(b)}_{dense_i}$ 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.

4.2 Solution

In the solution stage, we have two parallel regions and a sequential region (Eq. 3.18) between them. We can optimize the performance of these two parallel regions using the load-balance strategy explained in Chapter 3. This leaves us with Eq. 3.18 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 triangular matrix where d is at most the sum of all d_i explained in Section 4.1:

$$d \leq \sum_{i=1}^{t} d_i \tag{4.8}$$

since d_i values through partitions may contain duplicated columns. Solving the reduced system requires $\mathcal{O}(nnz(\widehat{S}) - d)$ 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,

$$\begin{gathered} \widehat{S} = I, \text{ when } r_i > k_i, \forall i \in \{1, 2, ..., t\} \\ I\widehat{x} = \widehat{g} \text{ from Eq. 3.9} \\ \widehat{x} = \widehat{g} \end{gathered} \tag{4.9}$$

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 $[r_i, k_i]$.

CHAPTER 5

NUMERICAL EXPERIMENTS

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 [69]. Hereafter, we refer to them as PSTRSV and MKL, respectively. We have obtained twenty real-world test matrices from the SuiteSparse Matrix Collection [70] that arise in variety of application areas and have a variety of dimensions/nonzeros (see Table 5.1 for properties and the application domains that they arise in).

As we have mentioned in Chapter 4, 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 [32, 33], Approximate Minimum Degree Permutation (AMD) [34], Column Permutation (ColPerm of Matlab R2018a), Nested Dissection Permutation (NDP) [35, 36], and Reverse Cuthill-McKee Ordering (RCM) [36]. After applying the permutation, we remove the strictly lower triangular part of the matrix to obtain U matrix. As explained in Section 2.1, 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 [71]. We repeat each run $1,000$ times

#	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 5.1: Properties of the test matrices.

and obtain the average of the required wallclock time. The required time to obtain the solution for PSTRSV and MKL are given for $t \in \{2, 4, 8, 10, 16, 20\}$ threads as well as the preprocessing times (for MKL this implies mkl_sparse_d_create_csr, mkl_sparse_set_sv_hint and mkl_sparse_optimize function calls) required by both in Appendix A.2. Preprocessing time excludes reordering time since it is common for both algorithms. Speed-ups obtained for each system are given in Appendix A.1. In the remaining parts of this chapter, we offer two perspectives built upon these results. First, we give a performance overview of the proposed algorithm against Intel MKL. Second, we present a case study to capture a detailed picture of the parallel performance for different matrix reordering algorithms.

5.1 Performance Overview

For performance overview, we present the number of test cases where the fastest solution is provided by a particular triangular solver in Figure 5.1. In addition, we give the best speed-up achieved by PSTRSV and MKL for all matrices in Figure 5.2. 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. For a more detailed breakdown of the speedups, we refer the reader to A.1. The final residuals obtained by PSTRSV are comparable with MKL.

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

$$s = \frac{min(runtime_{custom}, runtime_{MKL})}{runtime_{parallel}}. \tag{5.1}$$

In general, PSTRSV provides the best speedup for most of the test cases. This can be observed in Figure 5.1 where PSTRSV is better than others in 65% of the test cases on average for $t > 2$. Furthermore, in Figure 5.2, we present the best speed-ups 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. 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.

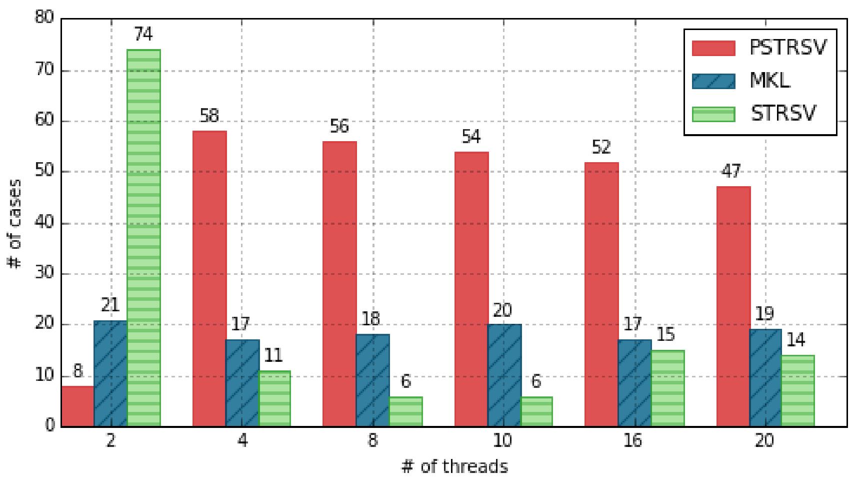

Figure 5.1: Overall performance comparison of the proposed solver, Intel MKL and the best sequential solver. Bars indicate the number of test cases where the given solver outperforms others. We ignore the test cases where we are unable to evaluate the performance due to memory constraints.

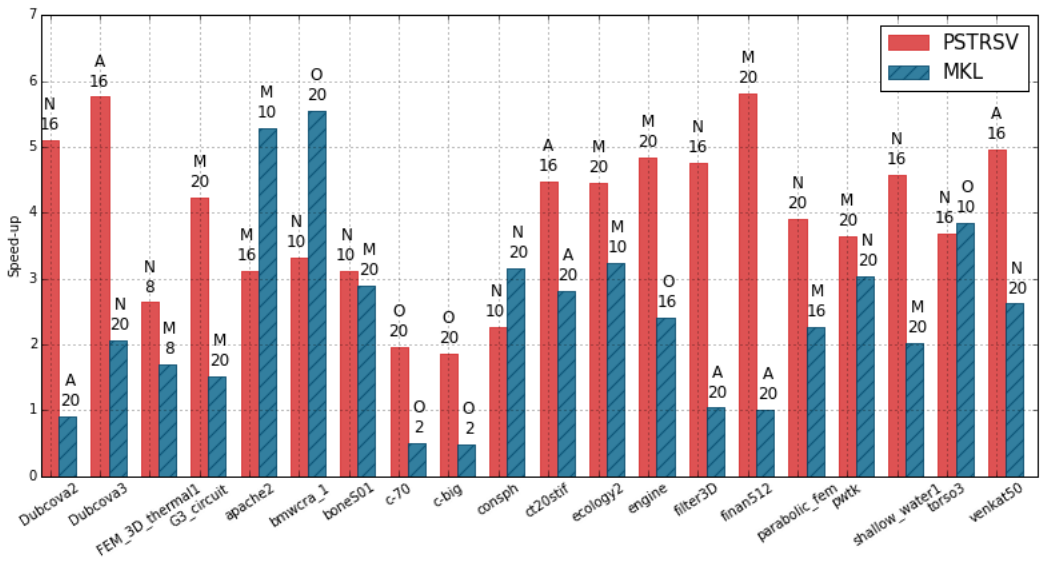

Figure 5.2: The highest speed-ups achieved by the proposed solver and Intel MKL solver. {R: RCM, C: ColPerm, N: NDP, M: METIS, A: AMD, O: ORIGINAL} symbols on bars indicate the matrix reordering algorithms which give the best result. The thread counts are placed under them.

t	PSTRSV				MKL			
	min	max	avg	std	min	max	avg	std
2	2.40	75.22	26.97	204.20	4.11	251.50	78.77	616.41
4	4.02	5995.39	875.11	10215.81	2.82	131.36	46.50	338.93
8	4.07	2988.42	576.13	5972.42	2.17	114.80	32.89	244.67
10	4.17	2756.16	495.46	5161.81	2.58	118.37	31.32	242.35
16	4.41	2961.46	372.92	4223.28	0.19	115.57	27.41	206.61
20	4.12	2219.22	327.21	3500.55	0.44	264.46	35.85	332.74

Table 5.2: Statistics of the preprocessing times of PSTRSV and MKL in milliseconds.

ColPerm and RCM, on the other hand, are not suitable for both PSTRSV and MKL.

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 $x$
```

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 5.2. Note that preprocessing stage of PSTRSV is parallel which is reflected as a decrease in the average preprocessing times in Table 5.2 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

t	min	max	avg	std
2	23	205	71.21	184.22
4	18	10572	944.44	14406.53
8	15	4772	378.24	5510.77
10	20	7525	317.20	7442.40
16	14	1517	226.86	2300.81
20	13	2229	209.18	2561.27

Table 5.3: Statistics regarding the required iterations by PSTRSV for amortization.

as explained in Section 4.1. 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 5.3, 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 5.2, average iteration count required for amortization drops as number of threads are increased (for $t > 2$). Although, overall MKL requires less preprocessing time than PSTRSV, it cannot amortize the lost time in $21/120$ test cases for any $t \in \{2, 4, 8, 10, 16, 20\}$, whereas PSTRSV cannot amortize the lost time only in $9/120$ test cases.

5.2 Case Study

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 5.4, 5.7, 5.10, 5.13, 5.16, and

5.19, and preprocessing times in Figures 5.5, 5.8, 5.11, 5.14, 5.17, and 5.20. Now, we look into those 6 cases where all reordering schemes work in more detail.

5.2.1 ct20stif

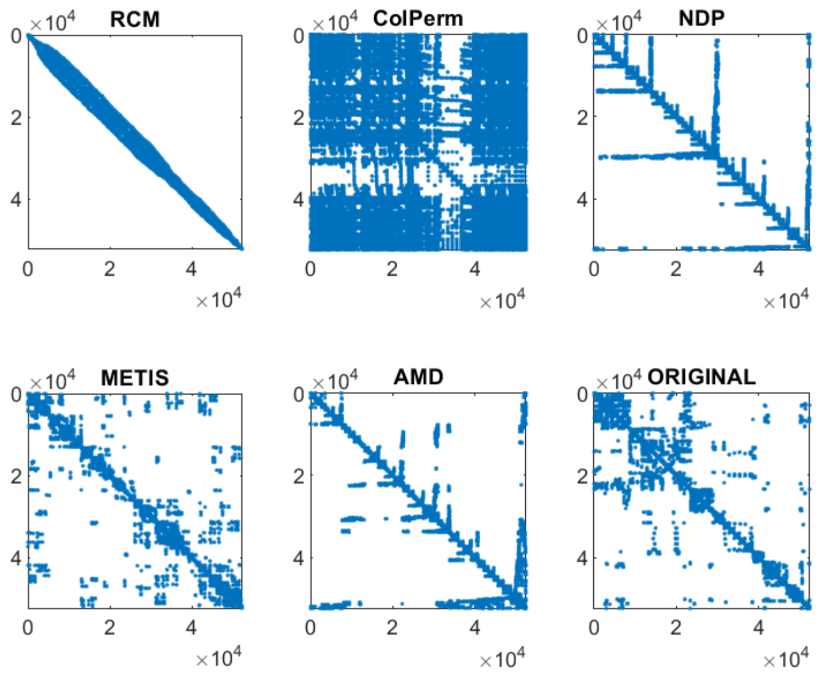

Figure 5.3: The illustration of *ct20stif* for different matrix reorderings.

ct20stif (Figure 5.3). According to Figure 5.4, PSTRSV outperforms MKL by obtaining a speed-up of $\sim 4\times$ by using NDP, METIS and AMD. However, MKL performs slightly better than PSTRSV when RCM, ColPerm, and ORIGINAL reorderings are employed, while the speed-up is poor (< 2). For preprocessing, MKL is faster than PSTRSV for $t > 2$. In Figure 5.5, it can be seen that PSTRSV benefits the most from METIS and NDP whereas MKL favors RCM and AMD. For both solvers, ColPerm causes poor preprocessing performance.

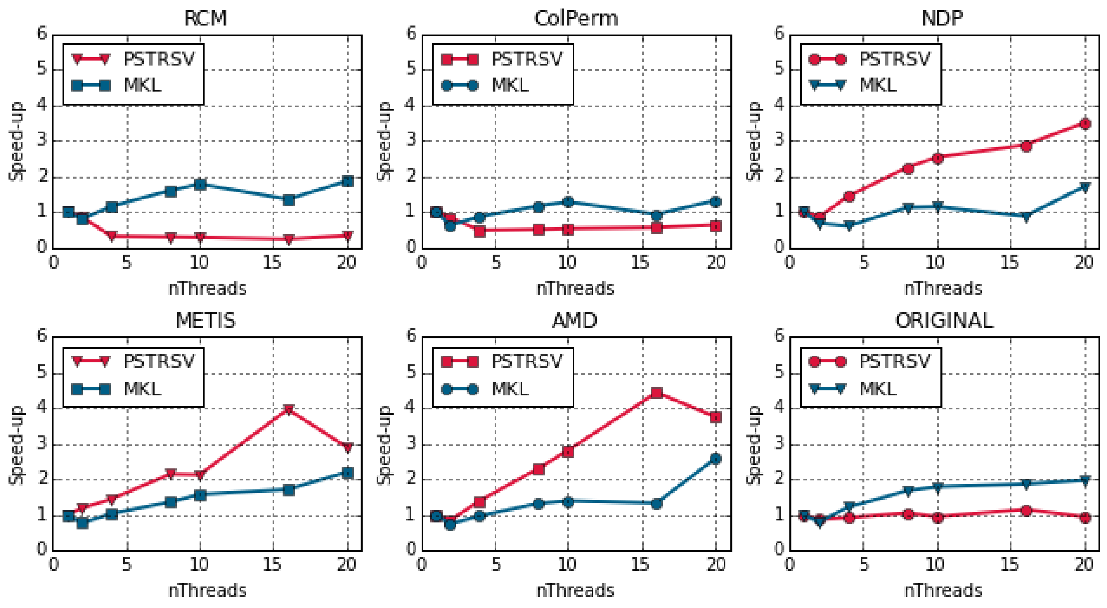

Figure 5.4: The speed-up comparison for *ct20stif*

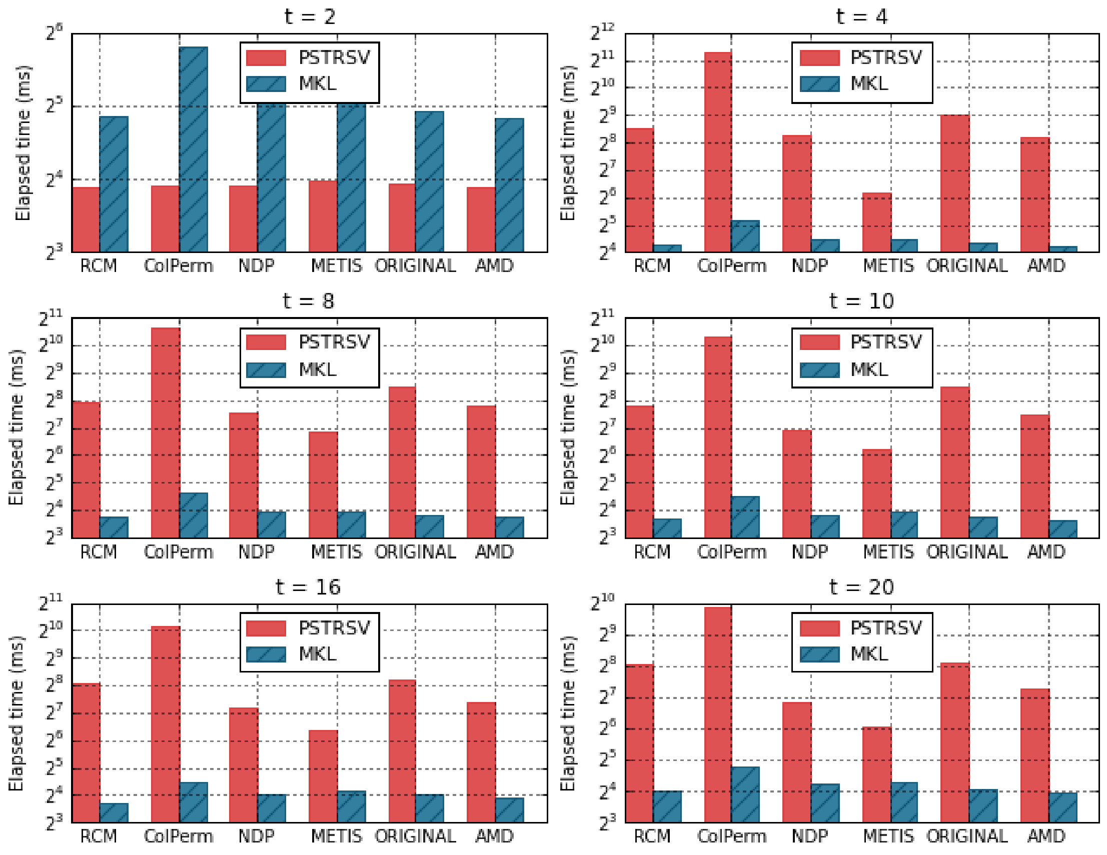

Figure 5.5: The preprocessing time comparison for *ct20stif*

5.2.2 FEM_3D_thermal1

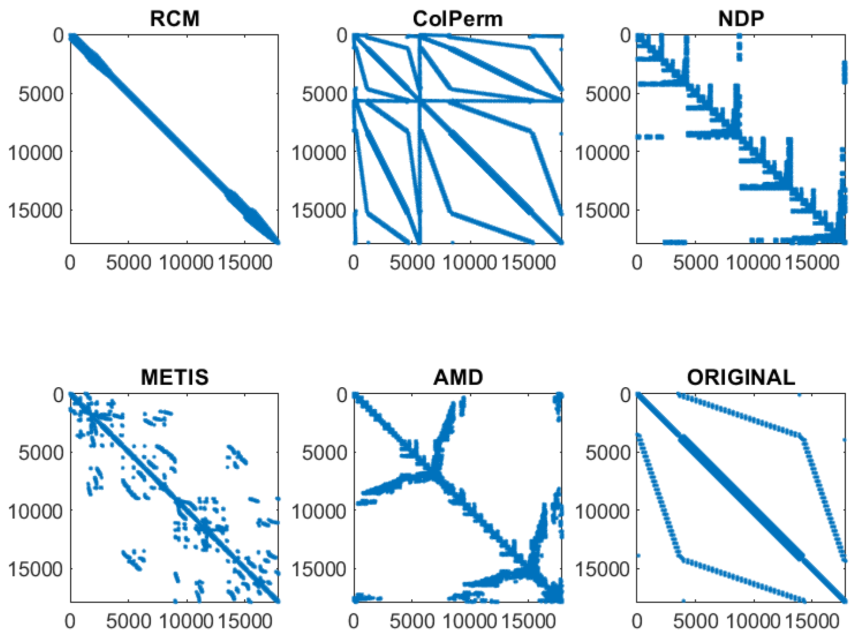

Figure 5.6: The illustration of *FEM_3D_thermal1* for different matrix reorderings.

FEM_3D_thermal1 (Figure 5.6). According to Figure 5.7, for all methods the speedup is poor. PSTRSV outperforms MKL only in NDP case by reaching $\sim 2.5\times$ speedup. For preprocessing, in Figure 5.8, PSTRSV outperforms MKL when NDP and ORIGINAL ordering are used for $t = 20$. For $t = 2$, PSTRSV again has a faster preprocessing phase. Nevertheless, PSTRSV benefits the most from METIS in all cases whereas RCM is the most suitable one for MKL. On the other hand, ColPerm and AMD are not suitable for PSTRSV.

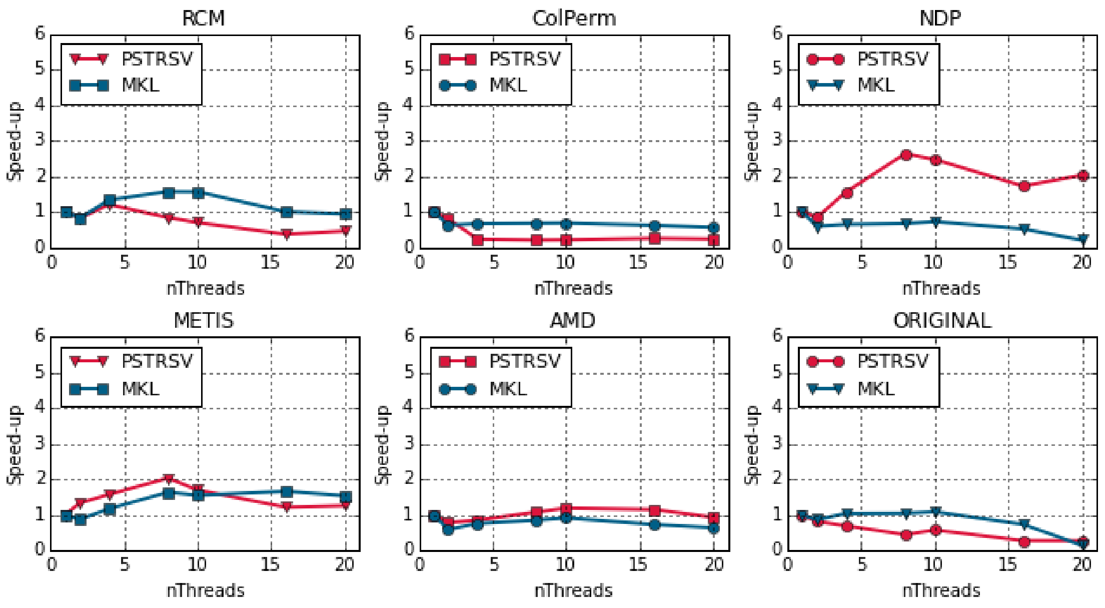

Figure 5.7: The speed-up comparison for *FEM_3D_thermal1*

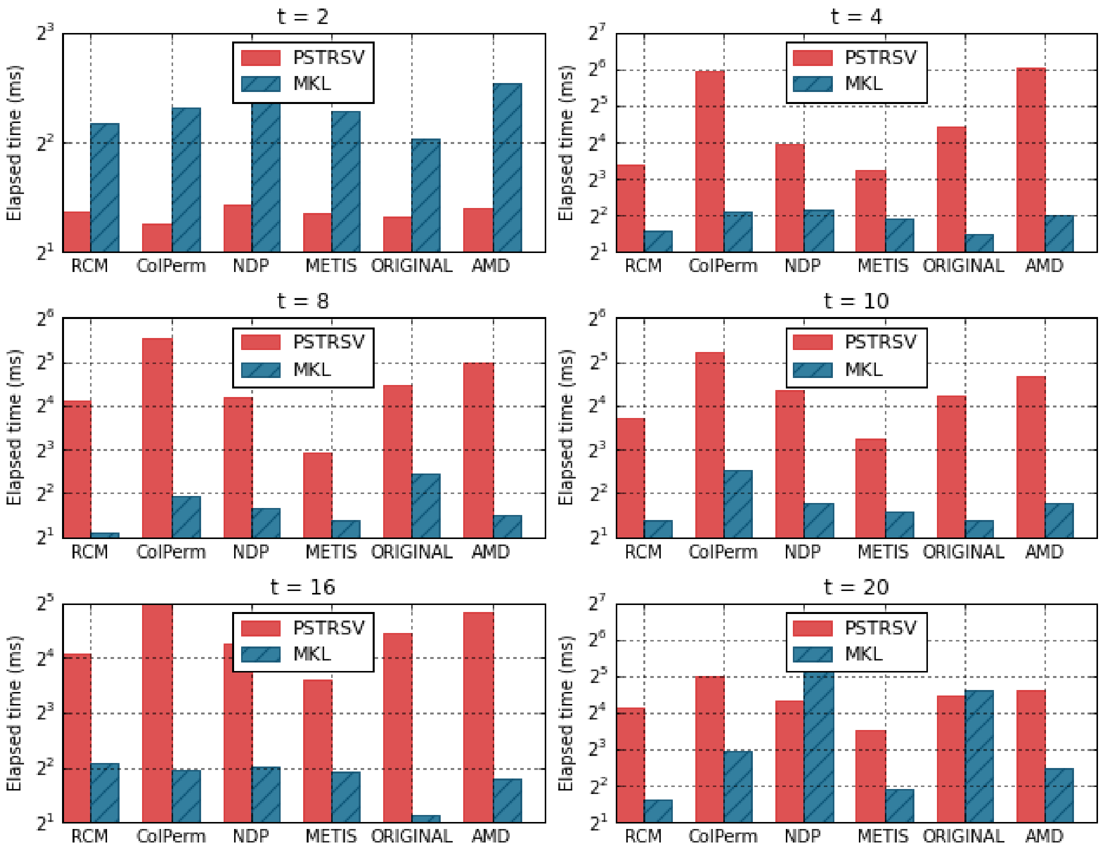

Figure 5.8: The preprocessing time comparison for *FEM_3D_thermal1*

5.2.3 finan512

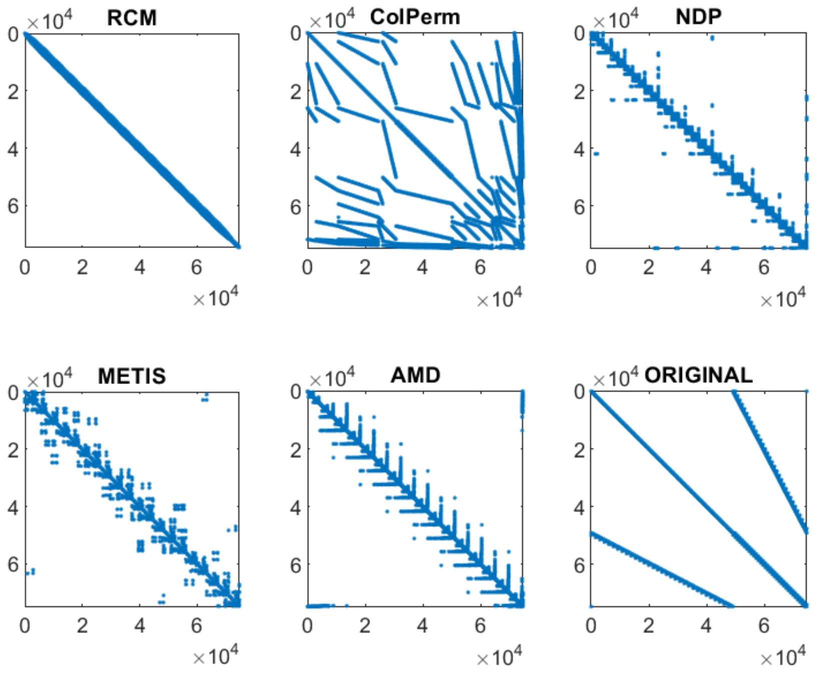

Figure 5.9: The illustration of *finan512* for different matrix reorderings.

finan512 (Figure 5.9). According to Figure 5.10, PSTRSV outperforms MKL in all cases except ColPerm, where both perform poorly. The best speed-up attained by PSTRSV is ~ 6. MKL consistently produces < 1 speed-up for all cases. For preprocessing, Figure 5.11 shows that PSTRSV requires lesser time than MKL when METIS is selected for $t \in \{2, 4, 20\}$. MKL outperforms PSTRSV in the rest of the cases for $t > 2$. As in Case 5.2.1, for both solvers, ColPerm deteriorates the preprocessing performance.

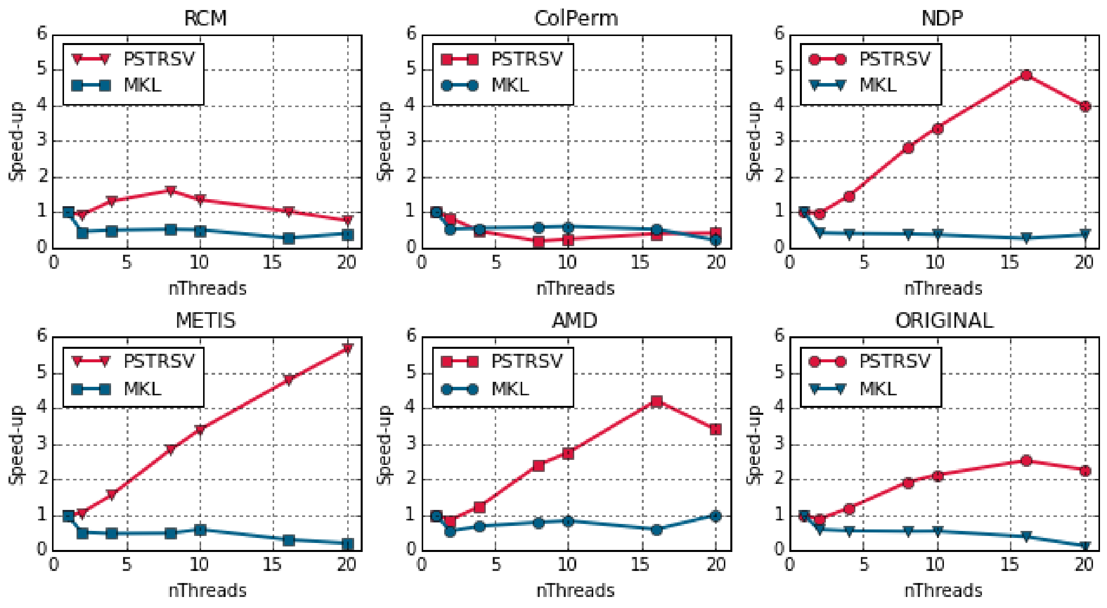

Figure 5.10: The speed-up comparison for *finan512*

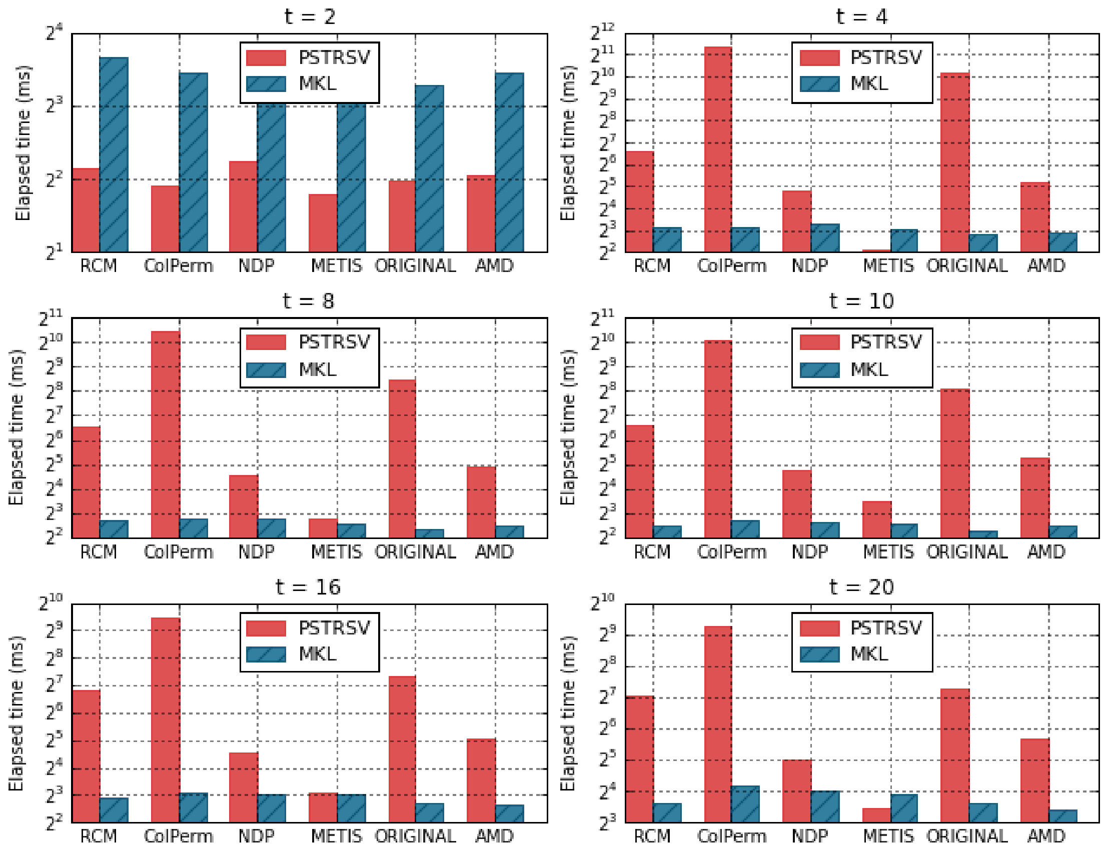

Figure 5.11: The preprocessing time comparison for *finan512*

5.2.4 pwtk

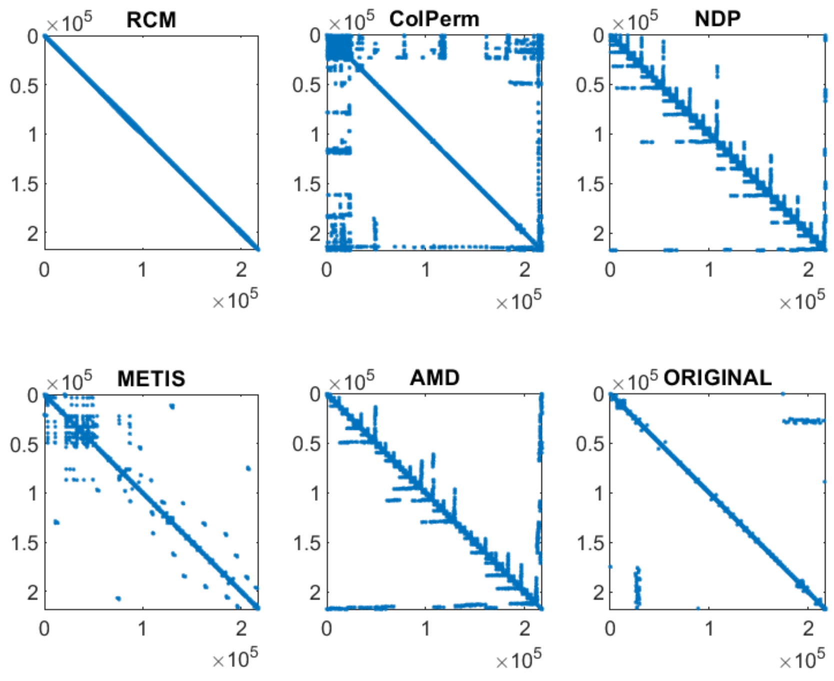

Figure 5.12: The illustration of *pwtk* for different matrix reorderings.

pwtk (Figure 5.12). According to Figure 5.13, PSTRSV outperforms MKL by reaching a speed-up of ~ 3 with NDP, METIS, and AMD. Poor parallelism with RCM results in worse performance than MKL which is able to reach $\sim 2\times$ speed-up. For preprocessing, MKL is faster than PSTRSV for $t > 2$. In Figure 5.14, it can be seen that PSTRSV benefits the most from METIS whereas MKL gets better performance with RCM, NDP, METIS and AMD. Again as in Cases 5.2.1 and 5.2.3, ColPerm is not suitable for both solvers.

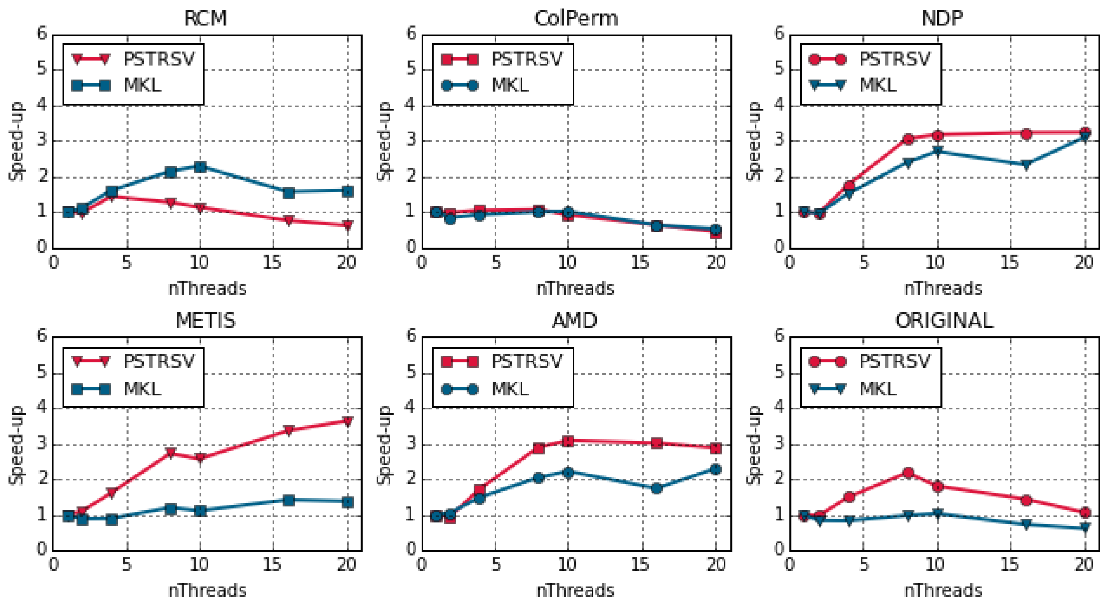

Figure 5.13: The speed-up comparison for *pwtk*

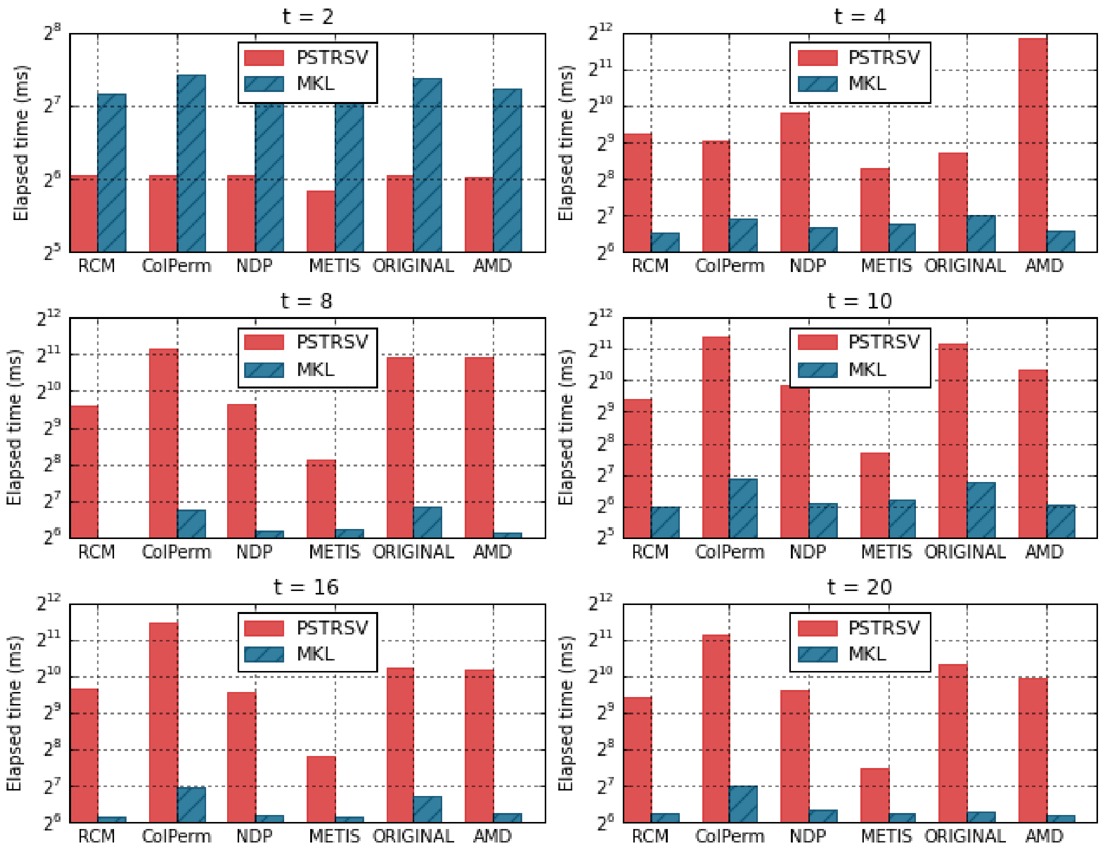

Figure 5.14: The preprocessing time comparison for *pwtk*

5.2.5 shallow_water1

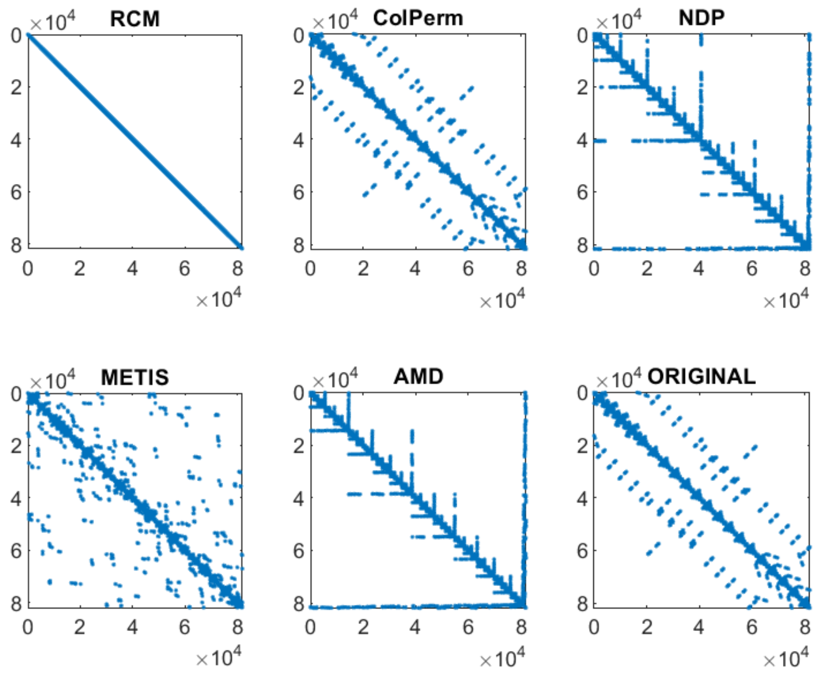

Figure 5.15: The illustration of *shallow_water1* for different matrix reorderings.

shallow_water1 (Figure 5.15). According to Figure 5.16, PSTRSV achieves a good speed-up regardless the reordering method. PSTRSV outperforms MKL in all cases by a factor of ~ 4. For preprocessing, MKL outperforms PSTRSV for $t > 2$. In Figure 5.17, we can see that ColPerm, unlike the other cases, results in comparable preprocessing performance with METIS for PSTRSV when $t = 20$. Nevertheless, METIS is the best performer in overall for PSTRSV whereas RCM, NDP and AMD are not suitable in this case. For MKL, RCM produces the best results for $t \in \{4, 8, 16\}$, but it performs poorly for $t = 20$. Both solvers benefit from ColPerm, NDP and METIS for $t = 20$.

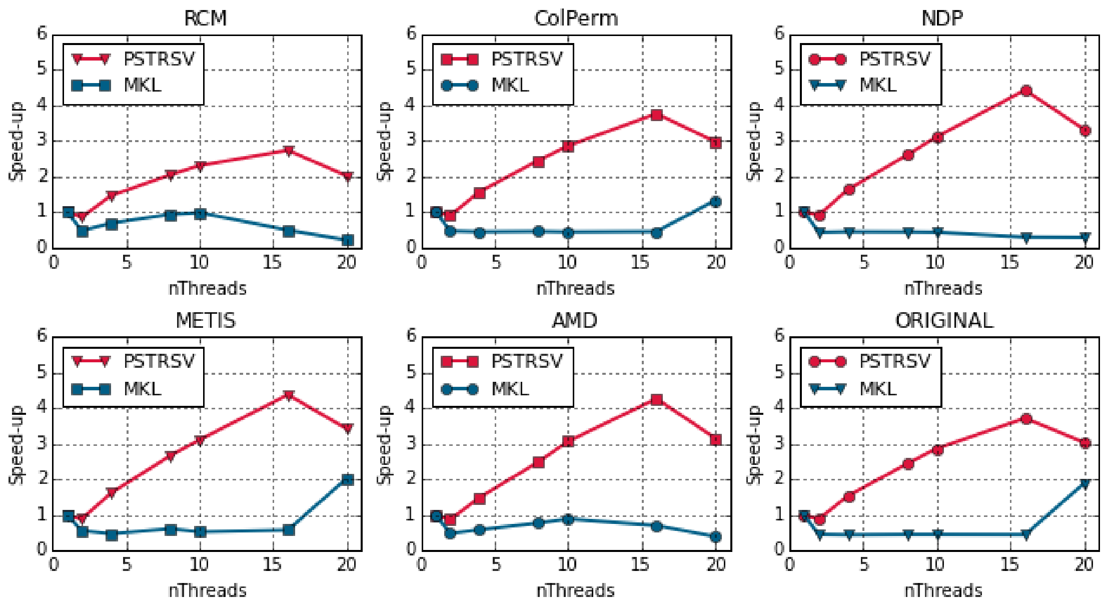

Figure 5.16: The speed-up comparison for *shallow_water1*

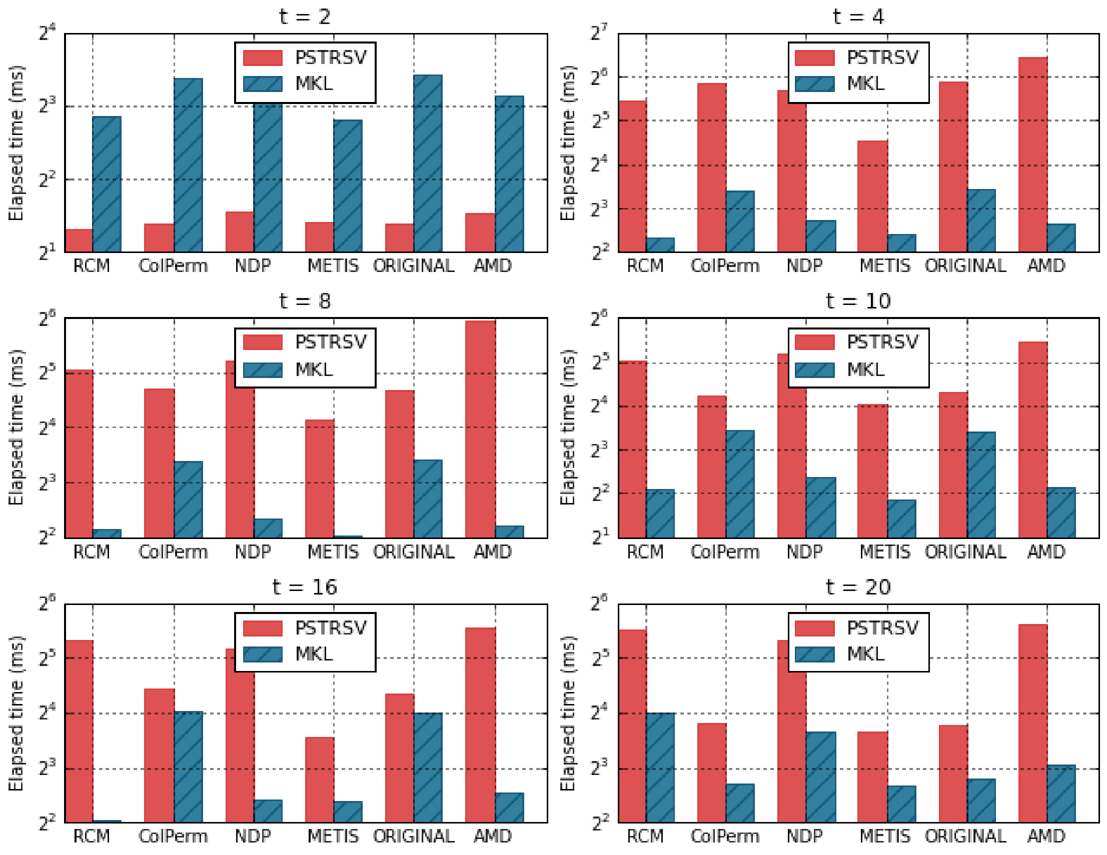

Figure 5.17: The preprocessing time comparison for *shallow_water1*

5.2.6 venkat50

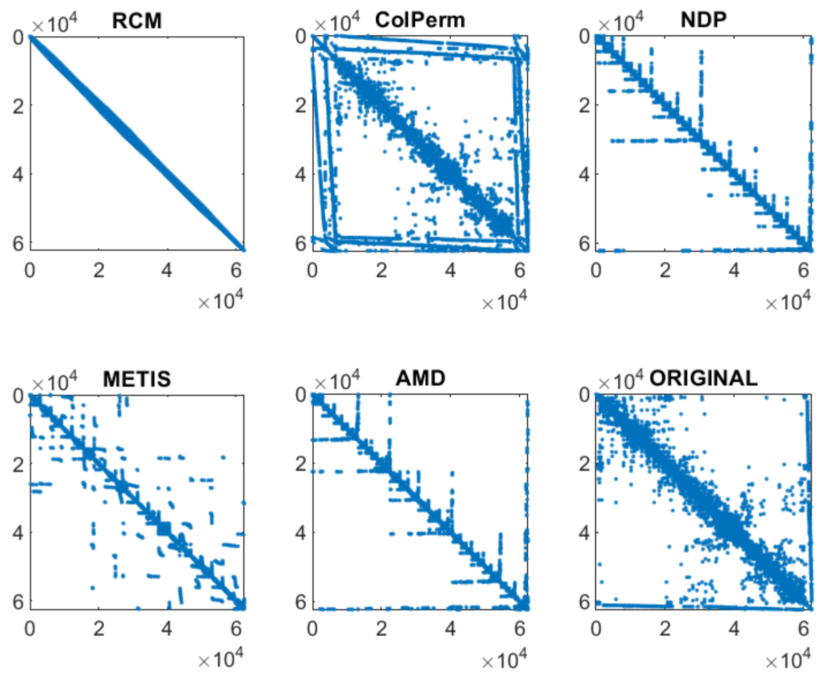

Figure 5.18: The illustration of *venkat50* for different matrix reorderings.

venkat50 (Figure 5.18). According to Figure 5.19, PSTRSV outperforms MKL by reaching at most $\sim 5\times$ speed-up for NDP, METIS, AMD, and ORIGINAL cases. Again, poor parallelism with RCM results in a worse performance than MKL which is able to reach $\sim 2\times$ speed-up. As in most cases, MKL outperforms PSTRSV in the preprocessing phase for $t > 2$. In Figure 5.20, METIS is the most suitable reordering for PSTRSV, and RCM is the most suitable one for MKL. As in Cases Cases 5.2.1, 5.2.3 and 5.2.4, ColPerm degrades the preprocessing performance for both solvers.

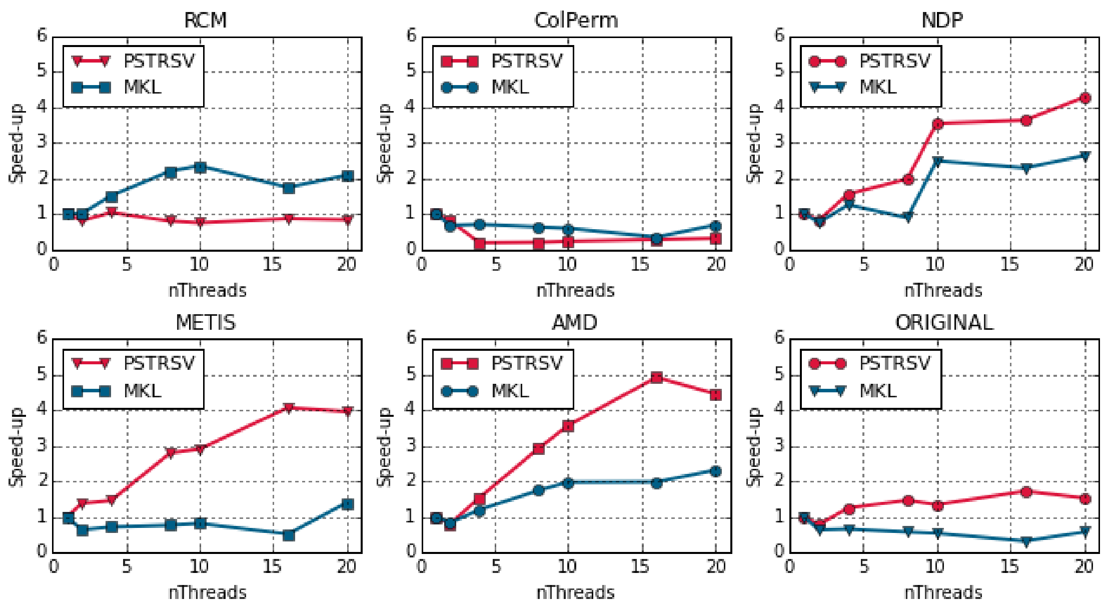

Figure 5.19: The speed-up comparison for *venkat50*

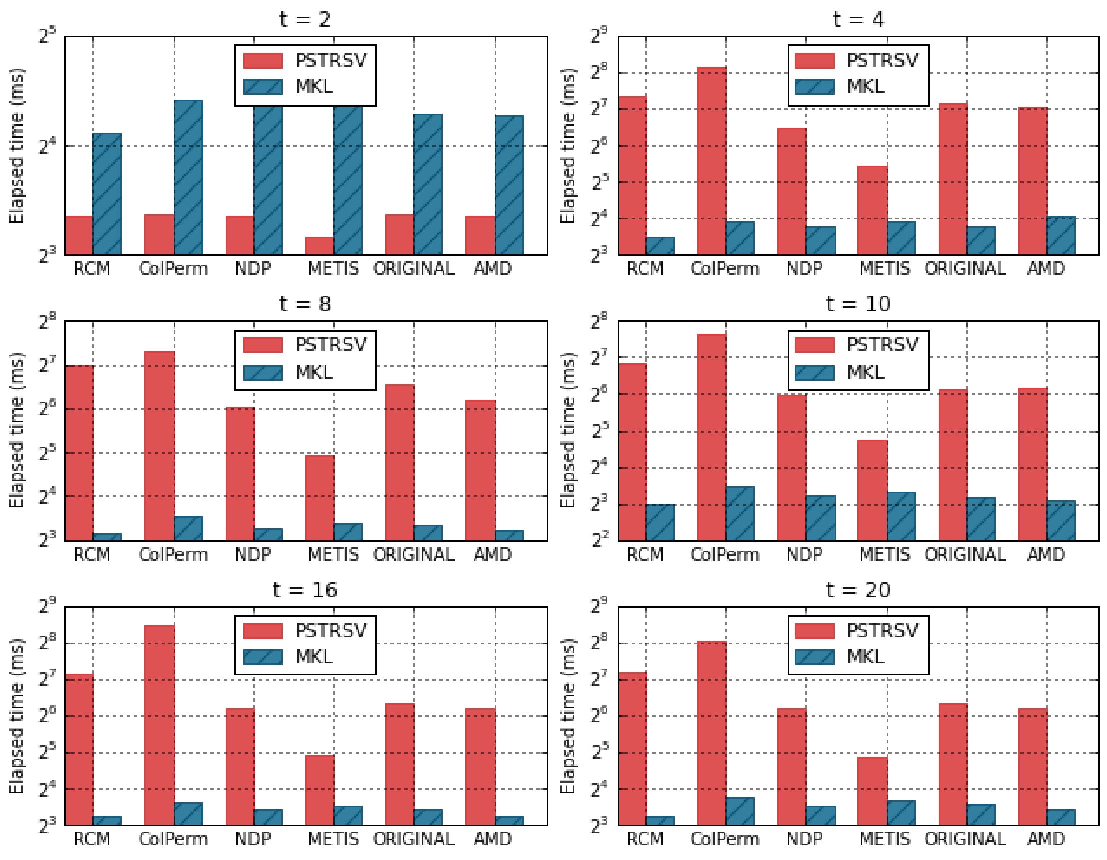

Figure 5.20: The preprocessing time comparison for *venkat50*

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, 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. According to the results, the proposed algorithm outperforms parallel sparse triangular solver of Intel MKL 2018 on a multicore arhitecture.

Several future work directions present themselves. First, a further study can be directed on the preprocessing performance of the proposed algorithm. In this work, there are some test cases where the proposed algorithm does not provide a solution due to the memory limitations we set, so a highly parallel approach with a reduced memory usage would solve this problem. Second, other matrix reordering frameworks such as PaToH [72] can be evaluated in terms of suitability for the proposed algorithm. Furthermore, we introduced the performance parameters of the proposed algorithm in Chapter 4. These parameters can be used to devise a specialized graph partitioning algorithm to improve the load-balance. Third, an MPI implementation of the proposed algorithm may prove useful for very large problems that are distributed among different processors.

REFERENCES

[1] D. Hysom and A. Pothen, “A scalable parallel algorithm for incomplete factor preconditioning,” *SIAM Journal on Scientific Computing*, vol. 22, no. 6, pp. 2194–2215, 2001.

[2] S. Hutchinson, J. Shadid, and R. Tuminaro, “Aztec user’s guide. version 1,” tech. rep., oct 1995.

[3] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, “Yale sparse matrix package i: The symmetric codes,” *International Journal for Numerical Methods in Engineering*, vol. 18, pp. 1145–1151, aug 1982.

[4] X. S. Li and J. W. Demmel, “Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 29, no. 2, pp. 110–140, 2003.

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

[6] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker, “Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation,” *Future Generation Computer Systems*, vol. 18, no. 1, pp. 69–78, 2001.

[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, and J. Koster, “A fully asynchronous multifrontal solver using distributed dynamic scheduling,” *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2001.

[9] T. A. Davis and I. S. Duff, “An unsymmetric-pattern multifrontal method for sparse lu factorization,” *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 1, pp. 140–158, 1997.

[10] S. Filippone and M. Colajanni, “Psblas: A library for parallel linear algebra computation on sparse matrices,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 4, pp. 527–550, 2000.

[11] M. Joshi, G. Karypis, V. Kumar, A. Gupta, and 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] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.

[13] “Caffe2: A new lightweight, modular, and scalable deep learning framework,” tech. rep., Facebook AI Research, USA, 2017.

[14] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” in *NIPS-W*, 2017.

[15] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan,

O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, E. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[16] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[17] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *International Journal of High Speed Computing*, vol. 1, no. 01, pp. 73–95, 1989.

[18] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM Journal on Scientific and Statistical Computing*, vol. 11, no. 1, pp. 123–144, 1990.

[19] R. Schreiber and W.-P. Tang, "Vectorizing the conjugate gradient method," *Proceedings of the Symposium on CYBER 205 Applications*, 1982.

[20] A. H. Sameh and R. P. Brent, "Solving triangular systems on a parallel computer," *SIAM Journal on Numerical Analysis*, vol. 14, no. 6, pp. 1101–1113, 1977.

[21] S.-C. Chen, D. J. Kuck, and A. H. Sameh, "Practical parallel band triangular system solvers," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 270–277, 1978.

[22] J. J. Dongarra and A. H. Sameh, "On some parallel banded system solvers," *Parallel Computing*, vol. 1, no. 3-4, pp. 223–235, 1984.

[23] E. Polizzi and A. H. Sameh, "A parallel hybrid banded system solver: the spike algorithm," *Parallel computing*, vol. 32, no. 2, pp. 177–194, 2006.

[24] E. Polizzi and A. Sameh, "Spike: A parallel environment for solving banded linear systems," *Computers & Fluids*, vol. 36, no. 1, pp. 113–120, 2007.

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

[26] O. Schenk, M. Manguoglu, A. Sameh, M. Christen, and M. Sathe, "Parallel scalable pde-constrained optimization: antenna identification in hyperthermia cancer treatment planning," *Computer Science-Research and Development*, vol. 23, no. 3-4, pp. 177–183, 2009.

[27] M. Manguoglu, "A domain-decomposing parallel sparse linear system solver," *Journal of Computational and Applied Mathematics*, vol. 236, no. 3, pp. 319–325, 2011.

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

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

[30] I. E. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, and A. H. Sameh, "A direct tridiagonal solver based on givens rotations for gpu architectures," *Parallel Computing*, vol. 49, pp. 101–116, 2015.

[31] K. Mendiratta and E. Polizzi, "A threaded spike algorithm for solving general banded systems," *Parallel Computing*, vol. 37, no. 12, pp. 733–741, 2011.

[32] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[33] G. Karypis and V. Kumar, “A parallel algorithm for multilevel graph partitioning and sparse matrix ordering,” *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 71–95, 1998.

[34] P. R. Amestoy, T. A. Davis, and I. S. Duff, “An approximate minimum degree ordering algorithm,” *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.

[35] A. George, “Nested dissection of a regular finite element mesh,” *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973.

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

[37] E. Totoni, M. T. Heath, and L. V. Kale, “Structure-adaptive parallel solution of sparse triangular linear systems,” *Parallel Computing*, vol. 40, no. 9, pp. 454–470, 2014.

[38] J. Mayer, “Parallel algorithms for solving linear systems with sparse triangular matrices,” *Computing*, vol. 86, no. 4, p. 291, 2009.

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

[40] E. Rothberg and A. Gupta, “Parallel iccg on a hierarchical memory multiprocessor — addressing the triangular solve bottleneck.,” *Parallel Computing*, vol. 18, no. 7, pp. 719 – 741, 1992.

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

[42] T. Iwashita, H. Nakashima, and 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*, pp. 474–483, IEEE, 2012.

[43] A. Pothen and F. L. Alvarado, "A fast reordering algorithm for parallel sparse triangular solution," *SIAM journal on scientific and statistical computing*, vol. 13, no. 2, pp. 645–653, 1992.

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

[45] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*, pp. 157–172, ACM, 1969.

[46] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *proc. IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[47] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu," tech. rep., NVIDIA Corp., Westford, MA, USA, 2011.

[48] Z. Chen, H. Liu, and B. Yang, "Parallel triangular solvers on gpu," *arXiv preprint arXiv:1606.00541*, 2016.

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

[50] X. Wang, W. Xue, W. Liu, and 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*, pp. 338–353, ACM, 2018.

[51] R. Li, "On parallel solution of sparse triangular linear systems in cuda," *arXiv preprint arXiv:1710.04985*, 2017.

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

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

[54] S. W. Hammond and R. Schreiber, “Efficient iccg on a shared memory multiprocessor,” *International Journal of High Speed Computing*, vol. 4, no. 01, pp. 1–21, 1992.

[55] M. Naumov, P. Castonguay, and J. Cohen, “Parallel graph coloring with applications to the incomplete-lu factorization on the gpu,” tech. rep., NVIDIA Corp., Westford, MA, USA, 2015.

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

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

[58] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and 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.

[59] B. Smith and H. Zhang, “Sparse triangular solves for ilu revisited: data layout crucial to better performance,” *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 386–391, 2011.

[60] E. Chow, H. Anzt, J. Scott, and J. Dongarra, “Using jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning,” *Journal of Parallel and Distributed Computing*, vol. 119, p. 219–230, 2018.

[61] H. Anzt, E. Chow, and J. Dongarra, “Iterative sparse triangular solves for preconditioning,” in *European Conference on Parallel Processing*, pp. 650–661, Springer, 2015.

[62] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on computers*, vol. 40, no. 5, pp. 603–612, 1991.

[63] M. Naumov, "Parallel incomplete-lu and cholesky factorization in the preconditioned iterative methods on the gpu," tech. rep., NVIDIA Corp., Westford, MA, USA, 2012.

[64] H. Liu, S. Yu, Z. Chen, B. Hsieh, and L. Shao, "Sparse matrix-vector multiplication on nvidia gpu," pp. 185–191, 2012.

[65] D. Brélaz, "New methods to color the vertices of a graph," *Communications of the ACM*, vol. 22, no. 4, pp. 251–256, 1979.

[66] D. P. Koester, S. Ranka, and G. Fox, "Parallel block-diagonal-bordered sparse linear solvers for electrical power system applications," in *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pp. 195–203, IEEE, 1993.

[67] T. Iwashita and M. Shimasaki, "Block red-black ordering: A new ordering strategy for parallelization of iccg method," *International Journal of Parallel Programming*, vol. 31, no. 1, pp. 55–75, 2003.

[68] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

[69] "Intel math kernel library. reference manual," tech. rep., Intel Corporation, Santa Clara, USA, 2018.

[70] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[71] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[72] U. V. Catalyürek and C. Aykanat, "Patoh: a multilevel hypergraph partitioning tool, version 3.0," *Bilkent University, Department of Computer Engineering, Ankara*, vol. 6533, 1999.

APPENDIX A

RESULTS OF ALL NUMERICAL EXPERIMENTS

A.1 Speed-up results

In this section, we present the computed speed-up s for each test matrix using the approach explained in Chapter 5. Due to memory constraints, the speed-up results are marked as "-" for the cases where we are not able to run both solvers.

A.1.1 Dubcova2

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.82	0.77	**0.83**	0.83	0.79	0.79
4	1.34	-	1.52	**1.53**	1.41	-
8	1.69	-	**2.94**	2.84	2.65	-
10	1.62	-	**3.46**	3.14	3.21	-
16	1.15	0.03	**5.11**	2.44	4.74	0.03
20	1.26	0.03	**4.04**	3.18	3.91	0.03

Table A.1: Speedup results of PSTRSV using different reoderings for *Dubcova2*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	**0.86**	0.49	0.50	0.59	0.53	0.58
4	**0.82**	-	0.49	0.63	0.63	-
8	**0.82**	-	0.50	0.62	0.75	-
10	**0.81**	-	0.49	0.53	0.78	-
16	**0.81**	0.55	0.37	0.26	0.67	0.59
20	0.81	0.33	0.26	0.34	**0.91**	0.64

Table A.2: Speedup results of MKL using different reoderings for *Dubcova2*

A.1.2 Dubcova3

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.85	0.88	0.85	**0.89**	0.85	0.81
4	1.11	-	**1.45**	1.26	1.39	-
8	1.12	-	**2.48**	1.52	2.28	-
10	1.04	-	**3.11**	2.48	2.90	-
16	0.73	-	2.91	1.81	**3.00**	-
20	0.54	-	3.02	1.50	**3.10**	-

Table A.3: Speedup results of PSTRSV using different reoderings for *Dubcova3*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.64	**0.85**	0.65	0.71	0.58	0.69
4	0.61	-	0.82	**0.84**	0.61	-
8	0.62	-	0.95	**0.99**	0.64	-
10	0.62	-	0.97	**1.00**	0.67	-
16	0.48	-	0.73	**0.93**	0.46	-
20	0.42	-	**1.82**	1.14	1.26	-

Table A.4: Speedup results of MKL using different reoderings for *Dubcova3*

A.1.3 FEM_3D_thermal1

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.83	0.80	0.85	**1.35**	0.77	0.84
4	1.20	0.23	1.57	**1.60**	0.84	0.69
8	0.84	0.20	**2.57**	2.06	1.07	0.45
10	0.69	0.21	**2.40**	1.74	1.22	0.58
16	0.37	0.25	**1.71**	1.21	1.18	0.28
20	0.45	0.23	**2.00**	1.26	0.94	0.27

Table A.5: Speedup results of PSTRSV using different reoderings for *FEM_3D_thermal1*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.83	0.62	0.60	**0.89**	0.58	0.89
4	**1.30**	0.67	0.64	1.14	0.76	1.03
8	1.55	0.67	0.65	**1.65**	0.84	1.06
10	1.55	0.67	0.71	**1.57**	0.92	1.10
16	1.00	0.62	0.52	**1.70**	0.73	0.75
20	0.94	0.55	0.20	**1.55**	0.65	0.14

Table A.6: Speedup results of MKL using different reoderings for *FEM_3D_thermal1*

A.1.4 G3_circuit

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.88	0.91	**0.99**	0.89	0.93	0.92
4	-	-	-	**1.63**	-	-
8	-	-	-	-	-	-
10	-	-	-	**3.44**	-	-
16	-	-	-	**3.32**	-	-
20	-	-	-	**4.23**	-	-

Table A.7: Speedup results of PSTRSV using different reoderings for *G3_circuit*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.30	**1.18**	0.46	1.07	0.40	1.00
4	-	-	-	**1.37**	-	-
8	-	-	-	-	-	-
10	-	-	-	**1.22**	-	-
16	-	-	-	**1.04**	-	-
20	-	-	-	**1.51**	-	-

Table A.8: Speedup results of MKL using different reoderings for *G3_circuit*

A.1.5 apache2

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.93	0.92	**0.97**	0.94	0.91	0.91
4	-	-	-	**1.54**	-	-
8	-	-	-	**1.49**	-	-
10	-	-	-	**1.85**	-	-
16	-	-	-	**3.06**	-	-
20	-	-	-	**1.31**	-	-

Table A.9: Speedup results of PSTRSV using different reoderings for *apache2*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.63	1.38	0.59	1.84	0.55	**1.85**
4	-	-	-	**2.97**	-	-
8	-	-	-	**4.40**	-	-
10	-	-	-	**5.26**	-	-
16	-	-	-	**4.68**	-	-
20	-	-	-	**4.55**	-	-

Table A.10: Speedup results of MKL using different reoderings for *apache2*

A.1.6 bmwcra_1

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.95	1.04	0.97	**1.24**	0.97	0.99
4	0.75	-	1.80	**2.04**	1.78	1.24
8	0.43	-	**3.03**	2.53	2.50	0.88
10	0.32	-	**3.31**	2.83	2.98	0.87
16	0.24	-	**1.73**	1.35	1.52	0.52
20	-	-	**1.67**	1.09	1.30	0.52

Table A.11: Speedup results of PSTRSV using different reoderings for *bmwcra_1*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	1.05	1.05	0.91	**1.22**	1.04	1.19
4	1.51	-	1.22	**1.93**	1.39	1.68
8	2.04	-	1.47	**3.34**	1.85	2.15
10	2.14	-	1.52	**3.88**	2.01	2.17
16	1.99	-	1.24	**4.37**	1.99	1.94
20	-	-	2.91	3.65	3.56	**5.55**

Table A.12: Speedup results of MKL using different reoderings for *bmwcra_1*

A.1.7 boneS01

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.92	0.94	0.91	**1.34**	0.92	0.95
4	0.29	-	**1.58**	0.89	1.57	0.34
8	0.24	-	**2.76**	1.64	2.75	0.19
10	0.23	-	3.10	2.28	**3.11**	0.18
16	0.21	-	**2.82**	0.60	2.63	-
20	0.17	-	**2.54**	1.35	2.52	-

Table A.13: Speedup results of PSTRSV using different reoderings for *boneS01*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.88	0.66	0.78	0.98	0.82	**0.99**
4	1.13	-	0.99	1.25	1.03	**1.44**
8	1.40	-	1.19	1.75	1.28	**2.03**
10	1.46	-	1.23	1.57	1.39	**2.38**
16	0.96	-	0.92	**1.54**	1.01	-
20	2.23	-	0.76	**2.90**	2.11	-

Table A.14: Speedup results of MKL using different reoderings for *boneS01*

A.1.8 c-70

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	-	-	-	-	-	**0.82**
4	-	-	-	-	-	**1.02**
8	-	-	-	-	-	**1.36**
10	-	-	-	-	-	**1.49**
16	-	-	-	-	-	**1.70**
20	-	-	-	-	-	**1.97**

Table A.15: Speedup results of PSTRSV using different reoderings for *c-70*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	-	-	-	-	-	**0.49**
4	-	-	-	-	-	**0.49**
8	-	-	-	-	-	**0.49**
10	-	-	-	-	-	**0.48**
16	-	-	-	-	-	**0.50**
20	-	-	-	-	-	**0.18**

Table A.16: Speedup results of MKL using different reoderings for *c-70*

A.1.9 c-big

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	-	-	-	-	-	**0.78**
4	-	-	-	-	-	**0.92**
8	-	-	-	-	-	**1.22**
10	-	-	-	-	-	**1.40**
16	-	-	-	-	-	**1.67**
20	-	-	-	-	-	**1.86**

Table A.17: Speedup results of PSTRSV using different reoderings for *c-big*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	-	-	-	-	-	**0.48**
4	-	-	-	-	-	**0.46**
8	-	-	-	-	-	**0.46**
10	-	-	-	-	-	**0.46**
16	-	-	-	-	-	**0.47**
20	-	-	-	-	-	**0.17**

Table A.18: Speedup results of MKL using different reoderings for *c-big*

A.1.10 consph

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.94	0.97	0.95	0.96	0.98	**1.08**
4	0.18	0.36	**1.55**	0.95	1.25	0.50
8	-	0.15	**1.71**	0.63	1.40	0.25
10	-	0.13	**2.20**	0.53	1.04	0.20
16	-	0.12	**1.39**	0.42	0.81	0.12
20	-	-	**1.39**	0.37	0.67	-

Table A.19: Speedup results of PSTRSV using different reoderings for *consph*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	**1.22**	1.11	0.92	1.08	0.90	0.89
4	**2.02**	1.49	1.23	1.80	1.02	0.89
8	-	2.02	1.49	**2.39**	1.17	0.98
10	-	2.25	1.56	**2.61**	1.16	0.96
16	-	1.71	1.21	**1.97**	0.83	0.62
20	-	-	**3.17**	2.90	2.68	-

Table A.20: Speedup results of MKL using different reoderings for *consph*

A.1.11 ct20stif

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.86	0.83	0.85	**1.18**	0.82	0.85
4	0.32	0.50	**1.44**	1.42	1.40	0.91
8	0.29	0.51	2.30	2.10	**2.48**	1.08
10	0.28	0.52	2.74	2.16	**2.77**	1.01
16	0.26	0.40	3.95	3.55	**4.36**	1.51
20	0.34	0.65	**3.62**	3.00	3.51	0.91

Table A.21: Speedup results of PSTRSV using different reoderings for *ct20stif*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.79	0.60	0.68	0.78	0.72	**0.79**
4	1.15	0.87	0.87	1.04	0.97	**1.23**
8	1.60	1.15	1.11	1.37	1.33	**1.68**
10	1.77	1.28	1.12	1.61	1.40	**1.79**
16	1.51	0.91	0.99	1.66	1.33	**1.97**
20	1.84	1.44	1.77	2.68	**2.82**	2.03

Table A.22: Speedup results of MKL using different reoderings for *ct20stif*

A.1.12 ecology2

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.85	**0.93**	0.92	0.93	0.91	0.93
4	1.10	1.24	1.39	**1.46**	-	1.32
8	1.29	-	2.34	**2.65**	-	-
10	1.31	-	-	**3.19**	-	-
16	1.26	-	2.46	**3.28**	-	-
20	1.17	-	-	**4.00**	-	-

Table A.23: Speedup results of PSTRSV using different reoderings for *ecology2*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.39	1.07	0.44	**2.04**	0.52	1.09
4	0.37	1.01	0.46	**1.35**	-	1.02
8	0.38	-	0.43	**1.54**	-	-
10	0.38	-	-	**3.21**	-	-
16	0.28	-	0.32	**1.35**	-	-
20	0.06	-	-	**2.22**	-	-

Table A.24: Speedup results of MKL using different reoderings for *ecology2*

A.1.13 engine

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	-	**0.92**	0.89	0.92	0.88	0.89
4	-	-	1.42	**1.55**	1.52	1.22
8	-	-	2.63	2.55	**2.65**	1.81
10	-	-	2.88	2.96	**2.97**	-
16	-	-	2.81	**2.91**	2.86	1.50
20	-	-	2.81	**3.07**	2.81	1.57

Table A.25: Speedup results of PSTRSV using different reoderings for *engine*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	-	0.51	0.62	0.64	**0.72**	0.64
4	-	-	0.84	0.97	0.97	**1.01**
8	-	-	1.12	1.24	1.31	**1.71**
10	-	-	1.18	1.26	**1.32**	-
16	-	-	0.93	0.99	1.14	**1.76**
20	-	-	1.93	1.63	**2.19**	1.80

Table A.26: Speedup results of MKL using different reoderings for *engine*

A.1.14 filter3D

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.87	0.81	0.87	**0.93**	0.84	0.87
4	0.84	-	**1.49**	1.27	1.36	0.64
8	0.59	-	**2.70**	2.32	2.14	0.17
10	0.47	-	3.14	**3.75**	2.77	0.11
16	0.30	-	**4.75**	2.88	3.25	0.08
20	0.24	-	**3.98**	2.02	2.52	0.07

Table A.27: Speedup results of PSTRSV using different reoderings for *filter3D*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	**0.68**	0.38	0.61	0.40	0.59	0.40
4	**0.84**	-	0.67	0.41	0.65	0.39
8	**0.96**	-	0.71	0.26	0.72	0.34
10	**0.96**	-	0.75	0.44	0.75	0.31
16	**0.69**	-	0.61	0.38	0.67	0.15
20	0.84	-	0.39	0.28	**1.04**	0.36

Table A.28: Speedup results of MKL using different reoderings for *filter3D*

A.1.15 finan512

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.92	0.83	0.95	**1.05**	0.85	0.88
4	1.29	0.45	1.45	**1.55**	1.24	1.17
8	1.62	0.18	2.78	**2.86**	2.42	1.91
10	1.34	0.24	3.36	**3.37**	2.70	2.10
16	1.02	0.38	**4.93**	4.85	4.20	2.52
20	0.75	0.40	3.89	**5.82**	3.50	2.25

Table A.29: Speedup results of PSTRSV using different reoderings for *finan512*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.45	0.52	0.41	0.51	0.54	**0.59**
4	0.48	0.54	0.39	0.47	**0.68**	0.55
8	0.51	0.56	0.37	0.48	**0.80**	0.54
10	0.50	0.59	0.35	0.59	**0.83**	0.54
16	0.26	0.50	0.25	0.30	**0.60**	0.39
20	0.39	0.20	0.35	0.20	**1.00**	0.13

Table A.30: Speedup results of MKL using different reoderings for *finan512*

A.1.16 parabolic_fem

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.92	0.86	**0.95**	0.91	0.91	0.85
4	1.36	-	**1.56**	1.42	-	-
8	2.05	-	**2.81**	2.64	-	-
10	2.27	-	**3.25**	2.99	2.97	-
16	2.51	-	2.60	2.42	**3.22**	-
20	2.25	-	**3.90**	2.53	2.43	-

Table A.31: Speedup results of PSTRSV using different reoderings for *parabolic_fem*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	**1.03**	0.61	0.48	0.60	0.47	0.60
4	**0.97**	-	0.49	0.84	-	-
8	0.96	-	0.49	**1.23**	-	-
10	0.96	-	0.49	**1.46**	0.57	-
16	0.95	-	0.41	**2.29**	0.43	-
20	**0.96**	-	0.16	0.94	0.84	-

Table A.32: Speedup results of MKL using different reoderings for *parabolic_fem*

A.1.17 pwtk

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.97	0.98	0.96	**1.10**	0.95	0.98
4	1.43	1.04	**1.76**	1.62	1.73	1.50
8	1.27	1.07	**3.05**	2.73	2.90	2.19
10	1.13	0.91	**3.16**	2.58	3.10	1.81
16	0.75	0.63	3.22	**3.37**	3.02	1.45
20	0.61	0.44	3.23	**3.64**	2.89	1.07

Table A.33: Speedup results of PSTRSV using different reoderings for *pwtk*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	**1.10**	0.83	0.96	0.89	1.06	0.85
4	**1.60**	0.92	1.51	0.90	1.49	0.84
8	2.14	1.00	**2.39**	1.21	2.05	0.98
10	2.29	1.01	**2.69**	1.11	2.23	1.04
16	1.57	0.62	**2.32**	1.42	1.75	0.73
20	1.60	0.51	**3.09**	1.39	2.30	0.62

Table A.34: Speedup results of MKL using different reoderings for *pwtk*

A.1.18 shallow_water1

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.85	0.90	**0.91**	0.89	0.88	0.89
4	1.46	1.56	**1.64**	1.61	1.49	1.51
8	2.00	2.48	2.62	**2.68**	2.52	2.48
10	2.33	2.85	3.06	3.11	**3.12**	2.85
16	2.80	3.80	**4.58**	4.54	4.42	3.80
20	2.00	3.00	3.24	**3.47**	3.12	3.00

Table A.35: Speedup results of PSTRSV using different reoderings for *shallow_water1*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.47	0.46	0.42	**0.57**	0.48	0.45
4	**0.67**	0.43	0.43	0.47	0.58	0.43
8	**0.93**	0.45	0.43	0.61	0.78	0.45
10	**0.98**	0.43	0.42	0.53	0.88	0.45
16	0.48	0.44	0.28	0.57	**0.71**	0.44
20	0.20	1.33	0.28	**2.03**	0.39	1.90

Table A.36: Speedup results of MKL using different reoderings for *shallow_water1*

A.1.19 torso3

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.83	0.83	**0.90**	0.85	0.76	0.83
4	0.32	-	**1.52**	1.49	-	1.08
8	-	-	**2.65**	1.02	-	0.98
10	-	-	**3.21**	1.25	-	0.92
16	-	-	**3.69**	1.43	-	0.41
20	-	-	**2.20**	1.09	-	0.38

Table A.37: Speedup results of PSTRSV using different reoderings for *torso3*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.57	0.96	0.53	1.02	0.60	**1.04**
4	0.59	-	0.48	1.54	-	**1.84**
8	-	-	0.45	2.02	-	**3.05**
10	-	-	0.43	2.27	-	**3.84**
16	-	-	0.28	1.79	-	**3.77**
20	-	-	0.25	2.47	-	**3.71**

Table A.38: Speedup results of MKL using different reoderings for *torso3*

A.1.20 venkat50

t	PSTRSV					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	0.80	0.79	0.82	**1.38**	0.81	0.78
4	1.03	0.18	**1.56**	1.45	1.53	1.25
8	0.79	0.20	2.00	2.77	**2.90**	1.47
10	0.74	0.23	**3.55**	2.89	3.53	1.36
16	0.86	0.28	3.67	4.11	**4.96**	1.73
20	0.83	0.31	4.23	3.93	**4.52**	1.54

Table A.39: Speedup results of PSTRSV using different reoderings for *venkat50*

t	MKL					
	RCM	ColPerm	NDP	METIS	AMD	ORIG
2	**1.00**	0.67	0.75	0.64	0.84	0.64
4	**1.50**	0.70	1.25	0.72	1.19	0.66
8	**2.20**	0.63	0.89	0.77	1.74	0.58
10	2.37	0.59	**2.50**	0.82	1.98	0.54
16	1.75	0.34	**2.29**	0.52	2.00	0.32
20	2.09	0.68	**2.62**	1.39	2.31	0.57

Table A.40: Speedup results of MKL using different reoderings for *venkat50*

A.2 Runtime results

In this section, we present the wall-clock times taken to perform the preprocessing and the solution phases of PSTRSV and MKL for each test case. Conformable with the Chapter 5, only the cases where both solvers are able to run are considered.

A.2.1 t = 2

Matrix	PSTRSV		MKL		STRSV
	Prep.	Sol.	Prep.	Sol.	
engine_NDP	28.7	**3.82**	77.39	5.41	3.93
engine_ColPerm	29.3	4.06	130.57	7.35	**3.69**
engine_ORIGINAL	29.7	3.94	95.37	5.46	**3.45**
engine_METIS	24.8	3.78	89.00	5.32	**3.38**
engine_AMD	29.3	3.91	77.08	4.81	**3.39**
consph_RCM	32.8	4.36	77.43	**3.37**	4.11
consph_ColPerm	33.0	4.28	85.60	**3.75**	4.11
consph_NDP	33.5	4.99	95.89	5.14	**4.79**
consph_METIS	33.3	4.42	76.61	**3.91**	4.35
consph_ORIGINAL	34.0	**3.89**	81.93	4.74	4.18
consph_AMD	33.3	4.83	89.64	5.26	**4.79**
bmwcra_1_RCM	59.8	8.64	136.74	**7.78**	8.16
bmwcra_1_ColPerm	59.2	8.19	207.78	**8.10**	8.48
bmwcra_1_NDP	60.0	9.19	161.76	9.78	**8.77**
bmwcra_1_METIS	50.8	**6.92**	137.08	7.03	8.62
bmwcra_1_ORIGINAL	59.8	8.66	142.68	**7.23**	8.54
bmwcra_1_AMD	59.9	9.09	151.51	**8.55**	8.73
shallow_water1_RCM	2.5	0.47	7.26	0.85	**0.42**
shallow_water1_ColPerm	2.6	0.61	10.99	1.20	**0.57**
shallow_water1_NDP	2.9	0.58	9.44	1.26	**0.55**
shallow_water1_METIS	2.7	0.63	7.44	0.99	**0.59**

shallow_water1_ORIGINAL	2.6	0.62	11.21	1.23	**0.57**
shallow_water1_AMD	2.9	0.58	8.89	1.06	**0.53**
FEM_3D_thermal1_RCM	2.5	0.35	4.59	0.35	**0.31**
FEM_3D_thermal1_ColPerm	2.4	0.35	4.94	0.45	**0.30**
FEM_3D_thermal1_NDP	2.7	0.40	6.96	0.57	**0.36**
FEM_3D_thermal1_METIS	2.5	**0.23**	5.42	0.35	0.34
FEM_3D_thermal1_ORIGINAL	2.5	0.37	4.33	0.35	**0.32**
FEM_3D_thermal1_AMD	2.6	0.39	6.01	0.52	**0.33**
c-70_ORIGINAL	3.9	0.72	19.88	1.20	**0.61**
parabolic_fem_RCM	24.5	7.75	73.41	**6.92**	7.14
parabolic_fem_ColPerm	23.4	5.11	93.90	7.27	**4.41**
parabolic_fem_NDP	32.8	5.96	91.51	11.66	**5.62**
parabolic_fem_METIS	22.2	4.88	80.76	7.42	**4.33**
parabolic_fem_ORIGINAL	23.6	5.24	92.28	7.34	**4.44**
parabolic_fem_AMD	29.9	5.29	83.63	10.33	**4.84**
c-big_ORIGINAL	15.7	3.57	84.51	5.82	**2.81**
venkat50_RCM	10.3	1.48	17.53	**1.19**	1.21
venkat50_ColPerm	10.5	1.36	21.44	1.62	**1.10**
venkat50_NDP	10.3	1.31	20.48	1.42	**1.10**
venkat50_METIS	9.0	**0.78**	23.94	1.70	1.10
venkat50_ORIGINAL	10.3	1.37	20.59	1.67	**1.09**
venkat50_AMD	10.2	1.37	19.50	1.32	**1.13**
boneS01_RCM	32.7	4.85	83.97	5.09	**4.34**
boneS01_ColPerm	32.9	5.00	179.99	7.04	**4.58**
boneS01_NDP	33.2	5.19	97.20	6.14	**4.65**
boneS01_METIS	34.4	**3.75**	88.34	5.23	5.12
boneS01_ORIGINAL	38.4	5.49	93.60	5.25	**5.06**
boneS01_AMD	33.3	4.95	83.36	5.54	**4.55**
ct20stif_RCM	14.9	1.63	29.08	1.78	**1.42**
ct20stif_ColPerm	14.9	1.80	57.19	2.49	**1.51**

ct20stif_NDP	14.9	1.68	34.50	2.11	**1.45**
ct20stif_METIS	15.9	**1.24**	35.03	1.86	1.50
ct20stif_ORIGINAL	15.3	1.70	31.56	1.83	**1.46**
ct20stif_AMD	15.1	1.73	29.74	1.97	**1.44**
finan512_RCM	4.5	0.66	13.96	1.37	**0.63**
finan512_ColPerm	3.8	0.63	11.78	1.00	**0.53**
finan512_NDP	4.8	0.76	14.92	1.77	**0.74**
finan512_METIS	3.5	**0.57**	10.50	1.17	0.64
finan512_ORIGINAL	3.9	0.69	10.77	1.03	**0.63**
finan512_AMD	4.2	0.71	10.89	1.11	**0.63**
torso3_RCM	27.8	4.49	74.75	6.53	**3.75**
torso3_ColPerm	26.0	4.95	76.26	4.28	**4.12**
torso3_NDP	31.4	5.43	105.85	9.22	**4.87**
torso3_METIS	27.0	4.83	65.43	**4.03**	4.15
torso3_ORIGINAL	27.3	4.95	76.44	**3.93**	4.12
torso3_AMD	33.5	5.99	106.73	7.64	**4.56**
Dubcova2_RCM	6.2	1.32	18.34	1.25	**1.10**
Dubcova2_ColPerm	6.1	1.01	22.11	1.58	**0.80**
Dubcova2_NDP	7.1	1.13	17.66	1.87	**0.97**
Dubcova2_METIS	6.4	1.03	20.46	1.45	**0.89**
Dubcova2_ORIGINAL	6.4	1.07	23.45	1.45	**0.85**
Dubcova2_AMD	7.1	1.12	17.24	1.65	**0.90**
Dubcova3_RCM	22.3	3.26	55.49	4.32	**2.80**
Dubcova3_ColPerm	20.4	3.04	57.44	3.13	**2.59**
Dubcova3_NDP	22.9	3.11	51.03	4.07	**2.64**
Dubcova3_METIS	21.9	3.09	76.08	3.89	**2.69**
Dubcova3_ORIGINAL	20.7	3.17	104.62	3.78	**2.51**
Dubcova3_AMD	22.9	3.22	49.36	4.71	**2.66**
G3_circuit_RCM	64.3	10.78	234.20	32.12	**9.50**
G3_circuit_ColPerm	58.0	21.21	203.92	**16.28**	19.24
G3_circuit_NDP	75.2	14.06	251.50	30.16	**13.98**

G3_circuit_METIS	57.5	22.52	163.60	**18.64**	19.11
G3_circuit_ORIGINAL	58.1	21.92	196.77	20.16	**20.13**
G3_circuit_AMD	69.5	12.83	248.66	29.49	**11.92**
pwtk_RCM	66.1	8.69	143.00	**7.65**	8.31
pwtk_ColPerm	66.2	8.91	171.29	10.49	**8.79**
pwtk_NDP	65.9	9.31	155.55	9.32	**8.98**
pwtk_METIS	57.5	**7.96**	165.46	9.81	8.78
pwtk_ORIGINAL	66.8	8.92	165.91	10.28	**8.76**
pwtk_AMD	65.4	9.12	150.39	**8.22**	8.75
apache2_RCM	35.6	5.48	94.48	8.01	**5.08**
apache2_ColPerm	33.5	10.41	104.08	**6.91**	9.57
apache2_NDP	41.0	6.92	129.28	11.36	**6.72**
apache2_METIS	32.9	10.12	84.33	**5.15**	9.28
apache2_ORIGINAL	33.5	10.57	89.95	**5.18**	9.58
apache2_AMD	36.8	6.36	119.38	10.65	**5.81**
ecology2_RCM	36.3	6.49	129.45	13.85	**5.58**
ecology2_ColPerm	37.2	13.43	133.54	**11.38**	12.46
ecology2_NDP	40.5	7.63	129.24	15.83	**7.18**
ecology2_METIS	36.3	13.28	91.10	**6.04**	12.32
ecology2_ORIGINAL	37.3	13.33	133.43	**11.37**	12.44
ecology2_AMD	40.8	7.61	106.62	13.09	**6.80**
filter3D_RCM	17.3	2.68	39.76	3.42	**2.34**
filter3D_ColPerm	17.6	2.60	96.35	5.51	**2.11**
filter3D_NDP	17.3	2.82	46.89	4.05	**2.47**
filter3D_METIS	15.2	2.73	62.86	6.32	**2.60**
filter3D_ORIGINAL	17.2	2.86	68.75	6.30	**2.53**
filter3D_AMD	17.8	2.87	43.38	4.11	**2.44**

Table A.41: The elapsed times of preprocessing and solution parts of the proposed algorithm and Intel MKL against the best sequential algorithm for different matrix reorderings. Measured in milliseconds. The number of threads is 2 for parallel solvers.

A.2.2 t = 4

Matrix	PSTRSV		MKL		STRSV
	Prep.	Sol.	Prep.	Sol.	
engine_NDP	661.9	**2.27**	51.27	3.99	3.93
engine_ORIGINAL	2241.3	**2.77**	73.24	3.41	3.45
engine_METIS	187.8	**2.27**	56.61	3.52	3.38
engine_AMD	1307.6	**2.32**	50.79	3.50	3.39
consph_RCM	1866.0	22.23	53.74	**2.03**	4.11
consph_ColPerm	2785.1	11.53	59.25	**2.78**	4.11
consph_NDP	1988.7	**3.06**	65.35	3.85	4.79
consph_METIS	765.4	4.53	54.21	**2.39**	4.35
consph_ORIGINAL	1149.5	8.54	60.97	4.74	**4.18**
consph_AMD	2806.7	**3.79**	59.00	4.64	4.79
bmwcra_1_RCM	1430.7	10.87	89.77	**5.44**	8.16
bmwcra_1_NDP	1844.3	**4.94**	103.81	7.28	8.77
bmwcra_1_METIS	115.3	**4.23**	92.70	4.46	8.62
bmwcra_1_ORIGINAL	4179.5	6.92	92.87	**5.09**	8.54
bmwcra_1_AMD	6019.8	**4.96**	103.57	6.35	8.73
shallow_water1_RCM	44.0	**0.28**	5.20	0.61	0.42
shallow_water1_ColPerm	58.3	**0.36**	11.38	1.29	0.57
shallow_water1_NDP	51.1	**0.33**	6.44	1.25	0.55
shallow_water1_METIS	26.2	**0.36**	5.38	1.23	0.59
shallow_water1_ORIGINAL	59.6	**0.37**	11.50	1.29	0.57
shallow_water1_AMD	87.8	**0.35**	6.56	0.90	0.53
FEM_3D_thermal1_RCM	10.3	0.25	3.04	**0.23**	0.31
FEM_3D_thermal1_ColPerm	62.5	1.29	4.49	0.45	**0.30**
FEM_3D_thermal1_NDP	16.1	**0.23**	4.71	0.56	0.36
FEM_3D_thermal1_METIS	9.7	**0.20**	3.77	0.28	0.34

FEM_3D_thermal1_ORIGINAL	21.2	0.48	2.92	**0.32**	0.32
FEM_3D_thermal1_AMD	65.8	0.38	4.12	0.42	**0.33**
c-70_ORIGINAL	4.0	**0.60**	21.06	1.24	0.61
parabolic_fem_RCM	446.3	**5.27**	70.28	7.40	7.14
parabolic_fem_NDP	516.0	**3.63**	61.89	11.54	5.62
parabolic_fem_METIS	413.6	**3.15**	54.92	5.30	4.33
c-big_ORIGINAL	16.1	3.02	81.89	6.07	**2.81**
venkat50_RCM	158.2	1.16	12.22	**0.80**	1.21
venkat50_ColPerm	284.9	6.01	16.16	1.58	**1.10**
venkat50_NDP	87.2	**0.70**	13.79	0.87	1.10
venkat50_METIS	43.0	**0.75**	15.27	1.52	1.10
venkat50_ORIGINAL	144.1	**0.87**	14.02	1.65	1.09
venkat50_AMD	131.8	**0.73**	12.81	0.94	1.13
rma10_AMD	307.6	1.59	0.88	1.66	**1.31**
boneS01_RCM	1563.1	15.27	54.82	**3.92**	4.49
boneS01_NDP	1385.0	**2.93**	61.97	4.68	4.68
boneS01_METIS	695.0	5.69	55.70	**4.06**	5.19
boneS01_ORIGINAL	1575.4	14.94	58.34	**3.52**	5.06
boneS01_AMD	2411.4	**2.90**	55.00	4.43	4.61
ct20stif_RCM	369.0	4.47	18.69	**1.23**	1.42
ct20stif_ColPerm	2520.1	2.99	35.40	1.73	**1.51**
ct20stif_NDP	304.1	**1.01**	22.83	1.66	1.45
ct20stif_METIS	70.5	**1.05**	23.56	1.43	1.50
ct20stif_ORIGINAL	531.7	1.61	20.14	**1.19**	1.46
ct20stif_AMD	298.1	**1.02**	19.13	1.47	1.44
finan512_RCM	97.4	**0.48**	8.76	1.29	0.63
finan512_ColPerm	2552.6	1.18	8.92	0.98	**0.53**
finan512_NDP	26.9	**0.51**	9.91	1.89	0.74
finan512_METIS	4.3	**0.40**	7.84	1.32	0.64
finan512_ORIGINAL	1174.9	**0.53**	7.02	1.13	0.63

finan512_AMD	36.5	**0.50**	7.59	0.91	0.63
torso3_RCM	2388.3	11.86	48.79	6.36	**3.75**
torso3_NDP	3000.1	**3.22**	69.01	10.11	4.87
torso3_METIS	850.7	2.75	45.36	**2.67**	4.15
torso3_ORIGINAL	2477.2	3.81	50.82	**2.23**	4.12
Dubcova2_RCM	47.7	**0.82**	18.22	1.34	1.10
Dubcova2_NDP	58.6	**0.63**	12.19	1.95	0.97
Dubcova2_METIS	43.0	**0.57**	14.26	1.38	0.89
Dubcova2_AMD	94.3	**0.64**	11.53	1.42	0.90
Dubcova3_RCM	182.7	**2.60**	51.86	4.59	2.80
Dubcova3_NDP	254.0	**1.86**	33.03	3.26	2.64
Dubcova3_METIS	245.9	**2.03**	45.98	3.22	2.69
Dubcova3_AMD	367.3	**1.98**	33.12	4.35	2.66
G3_circuit_METIS	1116.3	**12.00**	122.56	14.35	19.11
pwtk_RCM	611.8	5.92	92.12	**5.29**	8.31
pwtk_ColPerm	522.2	**8.36**	123.02	9.46	8.79
pwtk_NDP	905.8	**5.08**	101.59	5.91	8.98
pwtk_METIS	309.2	**5.38**	109.32	9.69	8.78
pwtk_ORIGINAL	423.0	**5.66**	131.36	10.12	8.76
pwtk_AMD	3714.4	**5.04**	95.76	5.85	8.75
apache2_METIS	982.4	6.18	56.88	**3.20**	9.28
ecology2_RCM	1444.2	**5.01**	120.04	14.81	5.58
ecology2_ColPerm	1540.4	**10.22**	128.47	12.16	12.46
ecology2_NDP	1463.3	**5.07**	96.30	15.46	7.18
ecology2_METIS	576.8	**8.52**	75.39	9.15	12.32
ecology2_ORIGINAL	1538.5	**9.55**	127.35	12.29	12.44
filter3D_RCM	299.0	2.79	25.67	2.79	**2.34**
filter3D_NDP	318.0	**1.65**	31.13	3.66	2.47
filter3D_METIS	110.4	**2.01**	41.57	6.31	2.60
filter3D_ORIGINAL	463.8	3.93	44.04	6.52	**2.53**

filter3D_AMD	426.7	**1.80**	28.86	3.73	2.44

Table A.42: The elapsed times of preprocessing and solution parts of the proposed algorithm and Intel MKL against the best sequential algorithm for different matrix reorderings. Measured in milliseconds. The number of threads is 4 for parallel solvers.

A.2.3 t = 8

Matrix	PSTRSV		MKL		STRSV
	Prep.	Sol.	Prep.	Sol.	
engine_NDP	395.4	**1.39**	37.92	3.02	3.93
engine_ORIGINAL	2044.1	**1.98**	49.70	2.04	3.45
engine_METIS	172.5	**1.25**	40.03	2.72	3.38
engine_AMD	621.4	**1.24**	35.55	2.58	3.39
consph_ColPerm	1565.0	28.11	42.21	**2.06**	4.11
consph_NDP	1216.0	**2.78**	46.74	3.18	4.79
consph_METIS	559.1	6.86	39.68	**1.80**	4.35
consph_ORIGINAL	787.9	17.04	42.07	4.35	**4.18**
consph_AMD	1494.5	**3.41**	41.30	4.08	4.79
bmwcra_1_RCM	895.1	18.82	65.02	**4.01**	8.16
bmwcra_1_NDP	1021.4	**2.92**	74.53	6.00	8.77
bmwcra_1_METIS	436.3	3.41	63.88	**2.58**	8.62
bmwcra_1_ORIGINAL	2092.8	9.74	66.15	**3.99**	8.54
bmwcra_1_AMD	3016.1	**3.54**	70.91	4.78	8.73
shallow_water1_RCM	33.5	**0.21**	4.33	0.45	0.42
shallow_water1_ColPerm	26.9	**0.23**	10.62	1.28	0.57
shallow_water1_NDP	37.1	**0.21**	5.36	1.27	0.55
shallow_water1_METIS	17.3	**0.22**	4.36	0.96	0.59
shallow_water1_ORIGINAL	26.4	**0.23**	11.03	1.28	0.57

shallow_water1_AMD	60.3	**0.21**	4.84	0.68	0.53
FEM_3D_thermal1_RCM	17.7	0.37	2.17	**0.20**	0.31
FEM_3D_thermal1_ColPerm	46.2	1.47	4.08	0.45	**0.30**
FEM_3D_thermal1_NDP	18.3	**0.14**	3.59	0.55	0.36
FEM_3D_thermal1_METIS	8.2	**0.16**	2.64	0.20	0.34
FEM_3D_thermal1_ORIGINAL	22.1	0.73	2.03	**0.31**	0.32
FEM_3D_thermal1_AMD	32.6	**0.30**	2.85	0.38	0.33
c-70_ORIGINAL	4.1	**0.45**	19.96	1.24	0.61
parabolic_fem_RCM	376.6	**3.48**	63.68	7.41	7.14
parabolic_fem_NDP	589.9	**2.00**	45.33	11.49	5.62
parabolic_fem_METIS	344.6	**1.68**	43.77	3.60	4.33
c-big_ORIGINAL	16.4	**2.27**	71.72	6.05	2.81
venkat50_RCM	128.8	1.53	8.96	**0.55**	1.21
venkat50_ColPerm	158.7	5.59	12.00	1.77	**1.10**
venkat50_NDP	97.6	**0.55**	17.99	1.23	1.10
venkat50_METIS	30.9	**0.40**	10.34	1.45	1.10
venkat50_ORIGINAL	92.3	**0.75**	9.90	1.90	1.09
venkat50_AMD	71.8	**0.39**	9.40	0.65	1.13
boneS01_RCM	1282.9	18.26	38.93	**3.17**	4.34
boneS01_NDP	672.7	**1.73**	45.78	3.92	4.65
boneS01_METIS	256.6	3.21	40.69	**2.92**	5.12
boneS01_ORIGINAL	1121.2	26.36	45.48	**2.51**	5.06
boneS01_AMD	1013.4	**1.82**	39.74	3.57	4.55
ct20stif_RCM	241.9	4.85	14.39	**0.89**	1.42
ct20stif_ColPerm	1521.4	2.95	24.98	**1.30**	1.51
ct20stif_NDP	187.2	**0.63**	15.60	1.31	1.45
ct20stif_METIS	114.9	**0.71**	15.94	1.09	1.50
ct20stif_ORIGINAL	362.2	1.35	14.01	**0.87**	1.46
ct20stif_AMD	217.9	**0.58**	13.49	1.08	1.44
finan512_RCM	94.5	**0.39**	6.81	1.24	0.63

finan512_ColPerm	1392.7	3.01	7.22	0.95	**0.53**
finan512_NDP	22.9	**0.27**	7.18	2.01	0.74
finan512_METIS	6.9	**0.22**	6.33	1.31	0.64
finan512_ORIGINAL	343.6	**0.33**	5.17	1.17	0.63
finan512_AMD	29.1	**0.26**	5.81	0.79	0.63
torso3_NDP	1580.0	**1.85**	52.09	10.93	4.87
torso3_METIS	638.6	4.04	31.84	**2.04**	4.15
torso3_ORIGINAL	1545.4	4.19	36.30	**1.35**	4.12
Dubcova2_RCM	38.6	**0.65**	17.37	1.34	1.10
Dubcova2_NDP	42.7	**0.33**	8.14	1.94	0.97
Dubcova2_METIS	28.6	**0.31**	10.41	1.43	0.89
Dubcova2_AMD	60.8	**0.34**	8.36	1.20	0.90
Dubcova3_RCM	156.3	**2.56**	50.34	4.57	2.80
Dubcova3_NDP	178.8	**1.08**	23.46	2.78	2.64
Dubcova3_METIS	264.4	**1.74**	30.43	2.71	2.69
Dubcova3_AMD	258.5	**1.03**	22.60	4.09	2.66
pwtk_RCM	762.5	6.67	64.59	**3.94**	8.31
pwtk_ColPerm	2301.0	**8.17**	106.55	8.75	8.79
pwtk_NDP	784.9	**2.93**	72.15	3.75	8.98
pwtk_METIS	277.5	**3.20**	75.44	7.23	8.78
pwtk_ORIGINAL	1923.5	**3.99**	114.37	8.89	8.76
pwtk_AMD	1906.4	**3.00**	69.43	4.24	8.75
apache2_METIS	897.6	6.28	46.60	**2.13**	9.28
ecology2_RCM	1176.1	**4.25**	115.14	14.79	5.58
ecology2_NDP	1270.1	**2.96**	78.19	16.63	7.18
ecology2_METIS	541.5	**4.72**	66.04	8.00	12.32
filter3D_RCM	215.2	4.01	18.16	2.45	**2.34**
filter3D_NDP	212.5	**0.92**	21.45	3.49	2.47
filter3D_METIS	70.0	**1.11**	33.72	9.95	2.60
filter3D_ORIGINAL	867.0	15.02	30.88	7.43	**2.53**

filter3D_AMD	397.5	**1.14**	19.64	3.38	2.44

Table A.43: The elapsed times of preprocessing and solution parts of the proposed algorithm and Intel MKL against the best sequential algorithm for different matrix reorderings. Measured in milliseconds. The number of threads is 8 for parallel solvers.

A.2.4 t = 10

Matrix	PSTRSV		MKL		STRSV
	Prep.	Sol.	Prep.	Sol.	
engine_NDP	341.6	**1.14**	35.76	2.86	3.93
engine_METIS	177.3	**1.16**	35.03	2.68	3.38
engine_AMD	515.7	**1.06**	34.50	2.41	3.39
consph_ColPerm	1106.0	32.35	39.00	**1.85**	4.11
consph_NDP	861.0	**2.16**	40.89	3.05	4.79
consph_METIS	468.4	8.13	34.24	**1.65**	4.35
consph_ORIGINAL	708.6	21.75	45.15	4.45	**4.18**
consph_AMD	1391.2	4.54	37.93	**4.08**	4.79
bmwcra_1_RCM	1059.0	25.28	60.59	**3.79**	8.16
bmwcra_1_NDP	908.4	**2.68**	72.00	5.81	8.77
bmwcra_1_METIS	390.2	3.03	60.70	**2.21**	8.62
bmwcra_1_ORIGINAL	1633.8	9.84	60.92	**3.95**	8.54
bmwcra_1_AMD	2134.8	**2.97**	66.91	4.41	8.73
shallow_water1_RCM	31.1	**0.18**	4.28	0.43	0.42
shallow_water1_ColPerm	19.1	**0.20**	11.35	1.32	0.57
shallow_water1_NDP	35.5	**0.18**	4.83	1.30	0.55
shallow_water1_METIS	16.3	**0.19**	3.77	1.12	0.59
shallow_water1_ORIGINAL	19.4	**0.20**	10.97	1.28	0.57

shallow_water1_AMD	43.1	**0.17**	4.49	0.60	0.53
FEM_3D_thermal1_RCM	13.1	0.45	2.67	**0.20**	0.31
FEM_3D_thermal1_ColPerm	36.8	1.43	5.88	0.45	**0.30**
FEM_3D_thermal1_NDP	20.7	**0.15**	3.26	0.51	0.36
FEM_3D_thermal1_METIS	10.7	**0.19**	2.90	0.21	0.34
FEM_3D_thermal1_ORIGINAL	19.1	0.59	2.66	**0.31**	0.32
FEM_3D_thermal1_AMD	24.3	**0.27**	3.91	0.36	0.33
c-70_ORIGINAL	4.1	**0.41**	18.91	1.26	0.61
parabolic_fem_RCM	368.0	**3.15**	62.49	7.45	7.14
parabolic_fem_NDP	748.5	**1.75**	42.25	11.62	5.62
parabolic_fem_METIS	302.0	**1.48**	39.16	3.04	4.33
parabolic_fem_AMD	1141.8	**1.66**	39.91	8.70	4.84
c-big_ORIGINAL	16.1	**1.99**	68.73	6.04	2.81
venkat50_RCM	114.5	1.63	7.79	**0.51**	1.21
venkat50_ColPerm	198.7	4.87	10.60	1.88	**1.10**
venkat50_NDP	62.7	**0.31**	9.66	0.44	1.10
venkat50_METIS	28.4	**0.38**	9.51	1.34	1.10
venkat50_ORIGINAL	70.9	**0.81**	8.95	2.03	1.09
venkat50_AMD	71.4	**0.32**	8.25	0.57	1.13
boneS01_RCM	1122.6	19.10	36.93	**3.04**	4.49
boneS01_NDP	573.0	**1.49**	41.17	3.76	4.68
boneS01_METIS	312.5	**2.23**	40.93	3.25	5.19
boneS01_ORIGINAL	954.7	27.94	41.10	**2.13**	5.06
boneS01_AMD	751.8	**1.48**	35.19	3.31	4.61
ct20stif_RCM	226.5	5.09	12.94	**0.80**	1.42
ct20stif_ColPerm	1265.9	2.88	21.87	**1.18**	1.51
ct20stif_NDP	118.8	**0.53**	13.91	1.29	1.45
ct20stif_METIS	74.8	**0.70**	16.28	0.94	1.50
ct20stif_ORIGINAL	360.3	1.46	13.48	**0.82**	1.46
ct20stif_AMD	180.8	**0.52**	12.47	1.03	1.44

finan512_RCM	98.5	**0.47**	6.01	1.27	0.63
finan512_ColPerm	1133.0	2.34	6.67	0.93	**0.53**
finan512_NDP	27.4	**0.22**	6.16	2.09	0.74
finan512_METIS	11.7	**0.19**	10.03	1.08	0.64
finan512_ORIGINAL	276.9	**0.30**	4.55	1.16	0.63
finan512_AMD	38.2	**0.23**	5.24	0.75	0.63
torso3_NDP	1206.2	**1.53**	47.82	11.40	4.87
torso3_METIS	575.8	3.33	29.75	**1.83**	4.15
torso3_ORIGINAL	1125.9	4.49	36.27	**1.07**	4.12
Dubcova2_RCM	39.9	**0.68**	17.79	1.36	1.10
Dubcova2_NDP	44.9	**0.28**	7.57	1.96	0.97
Dubcova2_METIS	34.3	**0.28**	9.60	1.65	0.89
Dubcova2_AMD	55.0	**0.28**	7.96	1.16	0.90
Dubcova3_RCM	150.5	**2.78**	48.82	4.57	2.80
Dubcova3_NDP	166.8	**0.85**	21.55	2.73	2.64
Dubcova3_METIS	201.7	**1.18**	27.78	2.71	2.69
Dubcova3_AMD	243.0	**0.85**	21.29	4.02	2.66
G3_circuit_METIS	1099.6	**5.64**	86.86	15.86	19.11
pwtk_RCM	693.0	7.51	63.24	**3.70**	8.31
pwtk_ColPerm	2756.2	9.57	116.75	**8.62**	8.79
pwtk_NDP	933.4	**2.82**	67.75	3.31	8.98
pwtk_METIS	208.5	**3.39**	72.50	7.84	8.78
pwtk_ORIGINAL	2361.3	**4.81**	108.38	8.42	8.76
pwtk_AMD	1330.9	**2.81**	65.53	3.91	8.75
apache2_METIS	708.8	5.07	45.20	**1.78**	9.28
ecology2_RCM	1106.3	**4.23**	117.68	14.84	5.58
ecology2_METIS	415.1	3.88	55.80	**3.82**	12.32
filter3D_RCM	222.8	5.02	17.04	2.44	**2.34**
filter3D_NDP	174.4	**0.79**	19.81	3.32	2.47
filter3D_METIS	46.9	**0.69**	24.94	5.90	2.60

filter3D_ORIGINAL	855.6	23.58	28.17	8.02	**2.53**
filter3D_AMD	376.0	**0.88**	18.10	3.26	2.44

Table A.44: The elapsed times of preprocessing and solution parts of the proposed algorithm and Intel MKL against the best sequential algorithm for different matrix reorderings. Measured in milliseconds. The number of threads is 10 for parallel solvers.

A.2.5 t = 16

Matrix	PSTRSV		MKL		STRSV
	Prep.	Sol.	Prep.	Sol.	
engine_NDP	236.4	**1.70**	29.73	3.16	3.93
engine_ORIGINAL	1633.2	2.13	38.27	**1.43**	3.45
engine_METIS	115.7	**1.35**	27.41	3.38	3.38
engine_AMD	320.0	**1.21**	32.41	2.73	3.39
consph_ColPerm	1013.4	34.94	39.69	**2.41**	4.11
consph_NDP	802.9	**3.38**	45.83	3.88	4.79
consph_METIS	512.2	10.40	41.42	**2.22**	4.35
consph_ORIGINAL	770.0	34.85	49.73	6.71	**4.18**
consph_AMD	918.3	5.87	43.40	5.78	**4.79**
bmwcra_1_RCM	733.9	33.62	46.21	**4.10**	8.16
bmwcra_1_NDP	675.6	**5.09**	53.74	7.09	8.77
bmwcra_1_METIS	407.8	6.36	46.72	**1.96**	8.62
bmwcra_1_ORIGINAL	838.9	16.57	47.55	**4.41**	8.54
bmwcra_1_AMD	1125.8	5.79	48.31	**4.43**	8.73
shallow_water1_RCM	28.7	**0.15**	4.89	0.87	0.42
shallow_water1_ColPerm	15.8	**0.15**	11.89	1.29	0.57
shallow_water1_NDP	26.6	**0.12**	5.43	1.93	0.55

shallow_water1_METIS	9.1	**0.13**	4.85	1.03	0.59
shallow_water1_ORIGINAL	15.3	**0.15**	12.03	1.29	0.57
shallow_water1_AMD	31.3	**0.12**	6.25	0.75	0.53
FEM_3D_thermal1_RCM	15.7	0.81	3.18	**0.30**	0.31
FEM_3D_thermal1_ColPerm	27.2	1.26	4.43	0.52	**0.30**
FEM_3D_thermal1_NDP	20.9	**0.21**	3.93	0.69	0.36
FEM_3D_thermal1_METIS	10.0	0.28	3.74	**0.20**	0.34
FEM_3D_thermal1_ORIGINAL	19.6	1.18	2.57	0.44	**0.32**
FEM_3D_thermal1_AMD	19.8	**0.28**	4.85	0.45	0.33
c-70_ORIGINAL	4.4	**0.37**	18.72	1.27	0.61
parabolic_fem_RCM	286.9	**2.84**	61.24	7.49	7.14
parabolic_fem_NDP	400.4	**2.16**	39.07	13.71	5.62
parabolic_fem_METIS	247.6	**1.82**	36.32	1.92	4.33
parabolic_fem_AMD	660.8	**1.50**	42.20	11.12	4.84
c-big_ORIGINAL	16.1	**1.65**	66.03	5.90	2.81
venkat50_RCM	105.4	1.40	10.32	**0.69**	1.21
venkat50_ColPerm	304.2	4.03	12.91	3.24	**1.10**
venkat50_NDP	63.4	**0.30**	9.34	0.48	1.10
venkat50_METIS	24.5	**0.27**	8.47	2.13	1.10
venkat50_ORIGINAL	60.4	**0.63**	11.62	3.44	1.09
venkat50_AMD	50.7	**0.23**	9.43	0.57	1.13
boneS01_RCM	1165.1	21.12	42.75	4.57	**4.49**
boneS01_NDP	534.5	**1.64**	46.63	5.02	4.68
boneS01_METIS	391.2	8.43	42.92	**3.28**	5.19
boneS01_AMD	682.3	**1.74**	41.75	4.52	4.61
ct20stif_RCM	194.9	5.45	12.25	**0.94**	1.42
ct20stif_ColPerm	950.2	3.75	23.59	1.63	**1.51**
ct20stif_NDP	84.7	**0.37**	13.97	1.48	1.45
ct20stif_METIS	57.6	**0.42**	13.21	0.90	1.50
ct20stif_ORIGINAL	225.2	0.97	14.16	**0.74**	1.46

ct20stif_AMD	95.2	**0.33**	12.10	1.08	1.44
finan512_RCM	71.7	**0.62**	6.55	2.42	0.63
finan512_ColPerm	594.4	1.40	7.46	1.05	**0.53**
finan512_NDP	18.8	**0.15**	7.14	2.95	0.74
finan512_METIS	7.8	**0.13**	6.71	2.09	0.64
finan512_ORIGINAL	122.4	**0.25**	5.58	1.62	0.63
finan512_AMD	23.8	**0.15**	5.98	1.05	0.63
torso3_NDP	696.5	**1.32**	44.83	17.10	4.87
torso3_METIS	262.6	2.91	26.94	**2.32**	4.15
torso3_ORIGINAL	729.4	10.01	30.24	**1.09**	4.12
Dubcova2_RCM	45.6	**0.95**	16.08	1.35	1.10
Dubcova2_ColPerm	614.8	30.05	10.93	1.51	**0.80**
Dubcova2_NDP	31.9	**0.19**	7.56	2.62	0.97
Dubcova2_METIS	27.5	**0.36**	8.29	3.37	0.89
Dubcova2_ORIGINAL	1453.0	27.63	12.79	1.45	**0.85**
Dubcova2_AMD	39.2	**0.19**	8.38	1.34	0.90
Dubcova3_RCM	137.5	4.46	46.99	4.56	**2.80**
Dubcova3_NDP	114.8	**0.53**	20.86	3.33	2.64
Dubcova3_METIS	146.4	**1.32**	21.26	2.97	2.69
Dubcova3_AMD	146.0	**0.49**	25.12	4.90	2.66
G3_circuit_METIS	912.4	**5.76**	102.83	18.38	19.11
pwtk_RCM	817.7	11.01	71.78	**5.29**	8.31
pwtk_ColPerm	2814.1	13.66	123.15	13.77	**8.79**
pwtk_NDP	763.6	**2.79**	72.35	3.87	8.98
pwtk_METIS	222.4	**2.61**	71.43	6.17	8.78
pwtk_ORIGINAL	1189.4	**6.05**	104.58	11.92	8.76
pwtk_AMD	1169.9	**2.84**	75.05	4.92	8.75
apache2_METIS	479.3	3.00	41.11	**1.88**	9.27
ecology2_RCM	858.5	**4.83**	113.25	14.75	5.58
ecology2_NDP	721.1	**3.40**	76.28	23.01	7.18

ecology2_METIS	428.8	**3.60**	54.11	5.85	12.32
coater2_ORIGINAL	5.9	0.24	0.19	0.24	**0.20**
filter3D_RCM	195.8	7.93	18.63	3.41	**2.34**
filter3D_NDP	136.2	**0.52**	18.06	4.06	2.47
filter3D_METIS	56.6	**0.90**	19.73	6.85	2.60
filter3D_ORIGINAL	769.4	32.07	26.29	16.54	**2.53**
filter3D_AMD	247.2	**0.75**	17.95	3.62	2.44

Table A.45: The elapsed times of preprocessing and solution parts of the proposed algorithm and Intel MKL against the best sequential algorithm for different matrix reorderings. Measured in milliseconds. The number of threads is 16 for parallel solvers.

A.2.6 t = 20

Matrix	PSTRSV		MKL		STRSV
	Prep.	Sol.	Prep.	Sol.	
engine_NDP	226.8	**1.43**	35.57	1.64	4.01
engine_ORIGINAL	1391.0	2.04	43.53	**1.45**	3.45
engine_METIS	105.0	**0.70**	28.63	1.53	3.38
engine_AMD	322.1	**0.81**	32.33	1.51	3.38
consph_NDP	429.9	3.41	39.61	**1.46**	4.75
consph_METIS	442.9	11.81	36.98	**1.50**	4.35
consph_AMD	594.0	6.31	36.22	**1.41**	4.74
bmwcra_1_NDP	653.8	5.26	62.98	**3.01**	8.77
bmwcra_1_METIS	419.2	7.94	54.27	**2.36**	8.62
bmwcra_1_ORIGINAL	857.7	16.27	49.09	**1.54**	8.54
bmwcra_1_AMD	981.6	6.71	54.41	**2.45**	8.73

shallow_water1_RCM	29.5	**0.21**	15.84	2.05	0.42
shallow_water1_ColPerm	12.1	**0.19**	7.65	0.43	0.57
shallow_water1_NDP	28.3	**0.17**	15.98	1.98	0.55
shallow_water1_METIS	10.9	**0.17**	6.61	0.29	0.59
shallow_water1_ORIGINAL	11.7	**0.19**	6.80	0.30	0.57
shallow_water1_AMD	32.5	**0.17**	8.27	1.37	0.53
FEM_3D_thermal1_RCM	13.8	0.69	3.19	0.33	**0.31**
FEM_3D_thermal1_ColPerm	24.9	1.30	6.92	0.55	**0.30**
FEM_3D_thermal1_NDP	16.0	**0.18**	6.67	1.81	0.36
FEM_3D_thermal1_METIS	9.8	0.27	4.23	**0.22**	0.34
FEM_3D_thermal1_ORIGINAL	17.8	1.18	153.93	2.28	**0.32**
FEM_3D_thermal1_AMD	20.9	0.35	4.09	0.51	**0.33**
c-70_ORIGINAL	4.1	**0.31**	26.47	3.44	0.61
parabolic_fem_RCM	281.7	**3.18**	59.91	7.40	7.14
parabolic_fem_NDP	405.1	**1.44**	95.54	34.70	5.62
parabolic_fem_METIS	244.7	**1.71**	57.17	4.61	4.33
parabolic_fem_AMD	644.8	**1.99**	55.95	5.74	4.84
c-big_ORIGINAL	16.7	**1.51**	102.03	17.03	2.81
venkat50_RCM	97.1	1.45	9.91	**0.58**	1.21
venkat50_ColPerm	237.1	3.58	15.57	1.62	**1.10**
venkat50_NDP	49.4	**0.26**	11.40	0.42	1.10
venkat50_METIS	23.1	**0.28**	11.04	0.79	1.10
venkat50_ORIGINAL	52.8	**0.71**	11.50	1.90	1.09
venkat50_AMD	52.3	**0.25**	11.38	0.49	1.13
rma10_AMD	118.3	5.61	0.44	1.69	**1.31**
boneS01_RCM	861.7	20.04	34.85	**1.79**	4.34
boneS01_NDP	478.9	**1.84**	61.65	6.16	4.68
boneS01_METIS	282.3	3.84	40.74	**1.79**	5.19
boneS01_AMD	599.6	**1.83**	46.71	2.19	4.61
ct20stif_RCM	207.8	4.14	15.81	**0.77**	1.42

ct20stif_ColPerm	793.5	2.34	26.01	**1.05**	1.51
ct20stif_NDP	80.0	**0.40**	16.47	0.82	1.45
ct20stif_METIS	57.1	**0.50**	12.46	0.56	1.50
ct20stif_ORIGINAL	162.7	1.60	13.64	**0.72**	1.46
ct20stif_AMD	94.9	**0.41**	13.93	0.51	1.44
finan512_RCM	89.7	0.84	11.50	1.62	**0.63**
finan512_ColPerm	437.5	1.33	14.91	2.60	**0.53**
finan512_NDP	23.3	**0.19**	13.64	2.12	0.74
finan512_METIS	9.5	**0.11**	26.28	3.19	0.64
finan512_ORIGINAL	129.6	**0.28**	12.80	4.81	0.63
finan512_AMD	41.3	**0.18**	9.11	0.63	0.63
torso3_NDP	924.3	**2.21**	80.57	19.63	4.87
torso3_METIS	258.7	3.82	29.77	**1.68**	4.15
torso3_ORIGINAL	594.8	10.76	31.40	**1.11**	4.12
Dubcova2_RCM	43.0	**0.87**	17.66	1.35	1.10
Dubcova2_ColPerm	527.2	28.68	18.09	2.39	**0.80**
Dubcova2_NDP	37.6	**0.24**	13.76	3.74	0.97
Dubcova2_METIS	22.4	**0.28**	16.33	2.64	0.89
Dubcova2_ORIGINAL	1091.1	28.21	21.68	1.32	**0.85**
Dubcova2_AMD	39.9	**0.23**	9.24	0.99	0.90
Dubcova3_RCM	130.9	3.83	47.61	4.49	**2.80**
Dubcova3_NDP	123.6	**0.66**	28.53	1.34	2.64
Dubcova3_METIS	126.3	**1.09**	27.15	2.58	2.69
Dubcova3_AMD	152.0	**0.64**	26.47	2.49	2.66
G3_circuit_METIS	953.2	**4.52**	110.75	12.64	19.11
pwtk_RCM	679.4	13.53	75.71	**5.19**	8.31
pwtk_ColPerm	2245.2	20.04	126.81	17.07	**8.79**
pwtk_NDP	784.7	**2.78**	81.35	2.91	8.98
pwtk_METIS	175.7	**2.41**	76.68	6.33	8.78
pwtk_ORIGINAL	1277.4	**8.19**	77.71	14.08	8.76

pwtk_AMD	988.5	**3.03**	73.70	3.80	8.75
apache2_METIS	860.5	7.06	57.48	**2.04**	9.28
ecology2_RCM	869.0	**5.19**	264.46	57.61	5.58
ecology2_METIS	273.2	**2.76**	53.84	3.82	12.32
filter3D_RCM	214.4	9.80	19.45	2.80	**2.34**
filter3D_NDP	118.4	**0.62**	34.86	6.33	2.47
filter3D_METIS	68.5	**1.29**	32.51	9.33	2.60
filter3D_ORIGINAL	822.6	38.08	39.09	7.10	**2.53**
filter3D_AMD	222.9	**0.97**	24.28	2.34	2.44

Table A.46: The elapsed times of preprocessing and solution parts of the proposed algorithm and Intel MKL against the best sequential algorithm for different matrix reorderings. Measured in milliseconds. The number of threads is 20 for parallel solvers.