A NEW MULTI-THREADED AND RECURSIVE DIRECT ALGORITHM FOR PARALLEL SOLUTION OF SPARSE LINEAR SYSTEMS

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

ERCAN SELÇUK BÖLÜKBAŞI

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

AUGUST 2013

Approval of the thesis:

A NEW MULTI-THREADED AND RECURSIVE DIRECT ALGORITHM FOR PARALLEL SOLUTION OF SPARSE LINEAR SYSTEMS

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

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering**

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

Examining Committee Members:

Prof. Dr. Sibel Tarı
Computer Engineering Department, METU

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

Prof. Dr. Bülent Karasözen
Mathematics Department, METU

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

Dr. Cevat Şener
Computer Engineering Department, METU

Date:

ABSTRACT

A NEW MULTI-THREADED AND RECURSIVE DIRECT ALGORITHM FOR PARALLEL SOLUTION OF SPARSE LINEAR SYSTEMS

Bölükbaşı, Ercan Selçuk
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Murat Manguoğlu

August 2013, 67 pages

Many of the science and engineering applications need to solve linear systems to model a real problem. Usually these linear systems have sparse coefficient matrices and thus require an effective solution of sparse linear systems which is usually the most time consuming operation. Circuit simulation, material science, power network analysis and computational fluid dynamics can be given as examples of these problems. With the introduction of multi-core processors, it became more important to solve sparse linear systems effectively in parallel. In this thesis, a new direct multi-threaded and recursive algorithm based on DS factorization to solve sparse linear systems will be introduced. The algorithmic challenges of this approach will be studied on matrices from different application domains. The advantages and disadvantages of variations of the algorithm on different matrices will be discussed. Specifically, we study the effects of changing number of threads, degree of diagonal dominance, the usage of sparse right hand side solution and various methods used to find the exact solution using the reduced system. Furthermore, comparisons will be made against a well known direct parallel sparse solver.

Keywords: Sparse Linear System, Parallel, Direct Solution, Recursion, Multithreading

ÖZ

SEYREK DOĞRUSAL SİSTEMLERİN PARALEL ÇÖZÜMÜ İÇİN ÇOK İZLEKLİ VE ÖZYİNELEMELİ YENİ BİR DOĞRUDAN ALGORİTMA

Bölükbaşı, Ercan Selçuk
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Murat Manguoğlu

Ağustos 2013 , 67 sayfa

Bilim ve mühendislik uygulamalarının çoğu, gerçek bir problemi modellemek için doğrusal sistemleri çözmeye ihtiyaç duyar. Bu doğrusal sistemler genellikle seyrek katsayılı matrislere sahiptirler ve bu nedenle çoğunlukla en çok zaman alan işlem olan seyrek doğrusal sistemlerin etkili bir çözümünü gerektirirler. Devre simülasyonu, malzeme bilimi, güç ağı analizi ve hesaplamalı akışkanlar dinamiği bu problemlere örnek olarak verilebilir. Çok çekirdekli işlemcilerin ortaya çıkması ile birlikte seyrek doğrusal sistemleri paralel olarak etkili bir biçimde çözmek önemli hale gelmiştir. Bu tez çalışmasında, seyrek doğrusal sistemleri çözmek için DS çarpanlara ayırma metoduna dayalı çok izlekli ve özyinelemeli yeni bir doğrudan algoritma tanıtılacaktır. Bu yaklaşımın algoritmik zorlukları farklı uygulama alanlarındaki matrisler üzerinde ele alınacaktır. Bu algoritmanın varyasyonlarının farklı matrislerdeki avantajları ve dezavantajları tartışılacaktır. Özel olarak; İzlek sayısında, çapraz baskınlıkta, seyrek sağ taraf çözümünün kullanımında ve küçültülmüş sistemi kullanarak kesin sonucu bulmak için kullanılan yöntemlerdeki değişikliklerin etkilerini ele alacağız. Buna ek olarak, iyi bilinen bir doğrudan paralel seyrek sistem çözücü ile de karşılaştırmalar yapılacaktır.

Anahtar Kelimeler: Seyrek Doğrusal Sistem, Paralel, Doğrudan Çözüm, Özyineleme, Çok İzlek

To My Family

ACKNOWLEDGMENTS

The author wishes to express his gratitude towards his supervisor Murat Manguoğlu for the guidance and the encouragement he provided.

The author would also like to thank to the examining committee members Prof. Dr. Sibel Tarı, Prof. Dr. Bülent Karasözen, Prof. Dr. Cevdet Aykanat and Dr. Cevat Şener for their valuable and constructive comments.

The HPC resources were provided by the Department of Computer Engineering, Middle East Technical University.

The Masters of Science education of the author was supported by The Scientific and Technological Research Council of Turkey (TUBİTAK) with Program No: 2210

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CERN	European Organization for Nuclear Research
Fermilab	Fermi National Accelerator Laboratory
LAPACK	Linear Algebra Package
MKL	Math Kernel Library
MPI	Message Passing Interface
MUMPS	A Multifrontal Massively Parallel Sparse Direct Solver
OpenMP	Open Multi-Processing
POSIX	Portable Operating System Interface for Unix
ScaLAPACK	Scalable Linear Algebra Package
UMFPACK	Unsymmetrix Multifrontal Sparse LU Factorization Package

CHAPTER 1

INTRODUCTION

The number of transistors in the processor's doubles every 18 months states the Moore's Law [1]. It is proven almost to be true by the developments of the last decades. And those improvements mostly reflected to the processor clock rates. Although this law would be limited to the possible minimum size of transistors as it will reach to the size of atoms, it can be said that it is expected to work for ten years more. However, mainly because of power dissipation issues, the tendency is now to produce processors with more cores rather than producing processors with higher clock rates. Therefore, even personal computers now have more cores instead of increase in clock rates. It is not surprising that high performance computers now have more and more nodes and cores by seeing the developments on personal computers.

The number of transistors is not the only number to increase over the years, but also problem data sizes and complexities. To have better and more realistic calculations we need models which resemble more and more to the real problems. Nonetheless, classical sequential methods are parallelized with pre-acceptance that they will perform well on parallel computing platforms. Nevertheless, they may not be the most efficient and scalable choice. Therefore in order to improve scalability of the linear solvers on multicore platforms, we need either parallelized versions of the classical linear solvers or novel methods that are tailored for today's parallel computing platforms.

Parallel computing is being used for a variety of areas, including applications in engineering and design, scientific applications, commercial applications and applications in computer sciences. Especially for areas that uses very large data, the linear systems constructed to represent them are also very large and it became almost obligatory to benefit from parallel computing [2, 3].

Those systems are actual models of real world problems and when they are represented as graphs, they usually have far less edges than a fully connected graph. Thus, they are sparse linear systems. In other words, in sparse matrices, there are more zero elements than other elements. Since these matrices have so many zero elements, they need to be treated specially to ensure no arithmetics is performed on zero elements.

An example of a sparsity structure of a matrix is given on Figure 1.1, with non-zero values shown with blue dots. It is a sparse matrix from University of Florida Matrix Collection [4], scircuit. It represents a circuit simulation problem. Only 0.0032795 percent of its entries are non-zero.

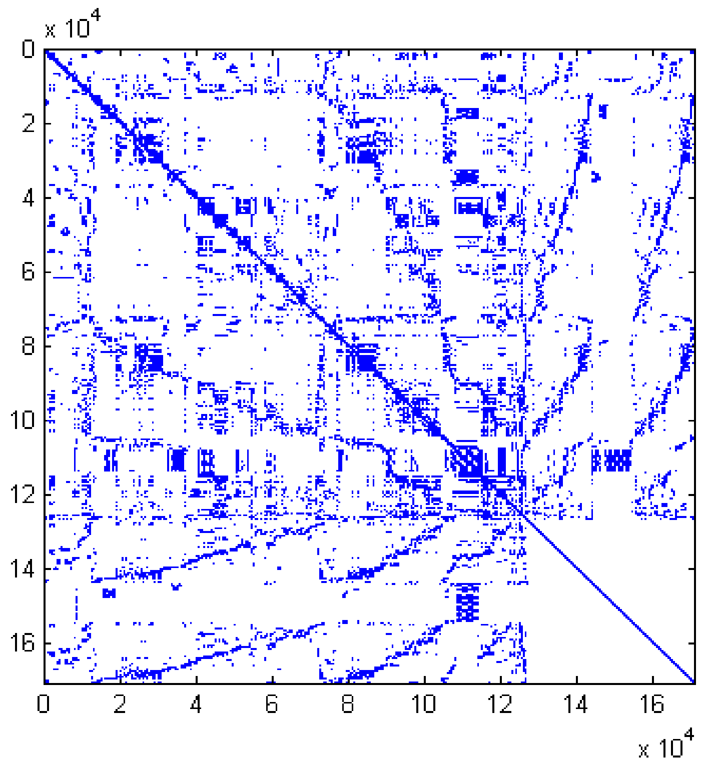

Figure 1.1: Matrix scircuit

To solve a linear system of equations, i.e., $Ax = f$ where x is unknown, a naive approach could be to calculate $x = A^{-1}f$. To solve the system with such approach, we need to find A^{-1} first. However, computing the inverse explicitly is not only computationally expensive but also numerically less stable. So, it is far more better to use methods to solve a linear system without explicitly computing the inverse of a matrix. Today, none of the linear system solvers computes the inverse of a matrix. Instead, they apply factorization on matrices. Our algorithm is based on one of them, namely the DS factorization. DS factorization is a new method to factorize matrices using the diagonal blocks.

Our algorithm parallely applies DS factorization on the initial matrix to be solved, in order to have an independent reduced system of the initial sparse linear system of equations. Moreover, we solve the reduce system applying the same algorithm recursively.

The main contribution of our new algorithm is that our algorithm is recursive, direct and multi-threaded [5].

The rest of this report is organized as follows: In Chapter 2, literature research about LU Factorization, SPIKE, and Domain Decomposing Parallel Sparse Linear Solver are explained. This chapter is followed by the chapter that explains Methods and Motivation including Domain Decomposition, Direct Solution, and Shared Memory and Recursion. In Chapter 4, our algorithm with all its variations is expressed. In Chapter 5, we show the programming and computing environment, and the experimental results. Finally, in Chapter 6, we conclude our work with a summary and comments on some possible future directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, background information about the solution of linear systems and factorization of matrices will be presented. Moreover, we will show two parallel algorithms that inspired our work.

2.1 LU Factorization

LU Factorization (LU Decomposition) is one of the classical algorithms that transforms the linear system of equations into a form that we can solve it by backward and forward substitutions. It can be described as the matrix form of Gaussian Elimination. The idea of LU Factorization is to transform the system into a product of an upper-triangular and a lower-triangular matrices (see figure 2.1), i.e., transforming $Ax = f$ to $LUx = f$, where $LU = A$.

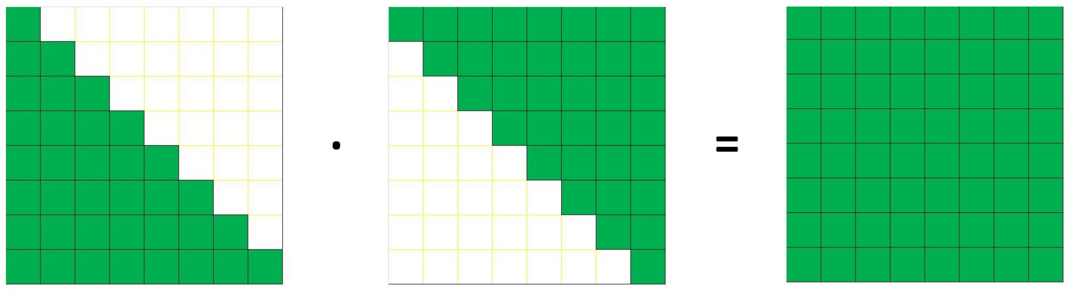

Figure 2.1: $LU = A$

For large condition numbers or singular matrices, Gaussian Elimination with partial or complete pivoting can be applied. The following is the classical dense LU Factorization without pivoting [6]. A basic pseudo code to solve a linear system is given on Algorithm 1.

Algorithm 1 LU Factorization (Doolittle Algorithm)

```
1: for $i \leftarrow 0, n-1$ do
2:     for $j \leftarrow i+1, n$ do
3:         $l_{i,j} \leftarrow a_{j,i}/a_{i,i}$
4:     end for
5:     for $j \leftarrow i+1, n$ do
6:         for $k \leftarrow i+1, n$ do
7:             $a_{j,k} = a_{j,k} + l_{j,i} \cdot a_{i,k}$
8:         end for
9:     end for
10: end for
```

Algorithm 1 overwrites the original matrix with diagonal and upper triangular values of U, and lower triangular values belonging to L. Since L is assumed to be a unit lower triangular matrix, the main diagonal of L known to consists of ones [6].

$Ax = f$ is now transformed into $LUx = f$ where L is a lower triangular matrix, U an upper triangular matrix, and $A = LU$.

After the factorization phase, forward and backward substitutions are applied.

Algorithm 2 Forward Substitution

```
1: $g_1 \leftarrow f_1/l_{1,1}$
2: for $i \leftarrow 2, n$ do
3:     $sum \leftarrow 0$
4:     for $j \leftarrow 1, i-1$ do
5:         $sum \leftarrow sum + l_{i,j} \cdot g_j$
6:     end for
7:     $g_i \leftarrow f_i - sum$
8: end for
```

In Algorithm 2, forward substitution is applied to find $g = L^{-1}f$. Since $LUx = f$, g is equal to LU. It is known that $Ux = g$. Thus, Algorithm 3 finds $x = U^{-1}g$ by backward substitution.

Algorithm 3 Backward Substitution

```
1: $x_n \leftarrow g_n/u_{n,n}$
2: for $i \leftarrow n-1, 1$ do
3:     $sum \leftarrow 0$
4:     for $j \leftarrow i+1, n$ do
5:         $sum \leftarrow sum + u_{i,i} \cdot x_j$
6:     end for
7:     $x_i \leftarrow g_i - sum$
8: end for
```

There are also a number of parallel algorithms that are using sparse LU factorization. An example is SuperLU which is a general purpose library for the direct solution of large, sparse, non-symmetric systems of linear equations in parallel and uses Gaussian elimination as its Kernel algorithm [7, 8].

2.2 SPIKE and Recursive SPIKE

The main idea of our work is based on the DS factorization of SPIKE [9, 10, 11]. The main difference is that SPIKE DS factorization is designed for banded linear systems, whereas our factorization is designed for sparse linear systems. SPIKE has two stages namely, pre-processing and post-processing, which are equivalent to factorization and solution stages in other solvers.

Figure 2.2 represents a banded sparse matrix for given number of partitions. On pre-processing phase, dark green cells are inside diagonal blocks whereas light green cells are outside.

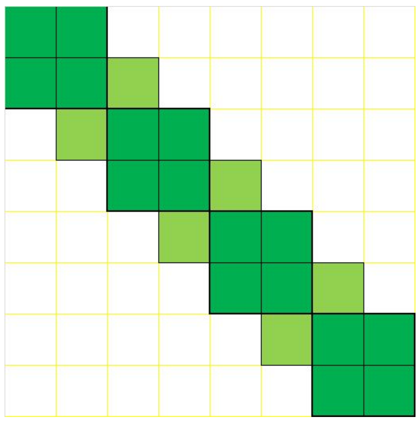

Figure 2.2: A Banded Sparse Matrix

A matrix after pre-processing is represented on Figure 2.3. Dark green cells inside the diagonal blocks are now Identity matrices and light green cells are dense matrices. Pre-processing stage does not work with the right hand side of the linear system. It has two phases: One to partition the linear system on different processes and one to simultaneously factorize the different partitions of the system on different processes to extract the reduced system with smaller size.

Post-processing, on the contrary, works with the right hand side of the system. This stage has also have two phases. First one is to solve the reduced system and the last

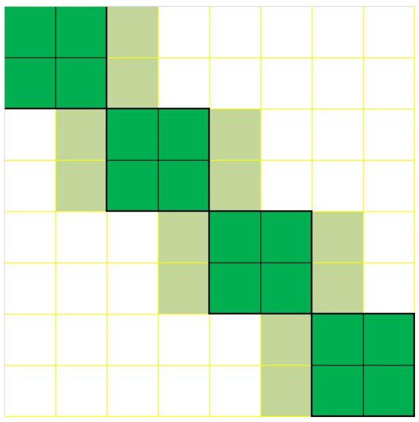

Figure 2.3: Matrix after pre-processing

phase to retrieve the solution of the initial system. Except the solution of the reduced system, all the steps of SPIKE are ideally parallel, i.e., does not need any data from any other processes. The size of the reduced system depends on the number of partitions and the bandwidth of the initial banded linear system.

There is also a recursive implementation of SPIKE [10]. The algorithm is applied recursively on S matrix, which has a structure that the block diagonals are identity and to the left and right block diagonals there are dense tall matrices. In our algorithm, however, the recursion is applied on the smaller reduced system.

2.3 Domain Decomposing Parallel Sparse Linear Solver

Domain Decomposing Parallel Sparse Linear Solver (DDPS) [12, 13] is a generalization of banded linear system solver SPIKE, for sparse linear systems. DDPS uses Algebraic domain decomposition. Like SPIKE, The algorithm is ideally parallel except the solution of the reduced system.

If DDPS is used as a solver for a preconditioned system in an iterative method, a dropping strategy is applied for speed-up. The error which is a result of dropping strategy is handled with iteration. However DDPS can be implemented as a direct solver if it is computed without a dropping strategy, and with exact computation of LU factorization of the initial matrix and exact solution of the reduced system. DDPS is designed to be used as a hybrid (direct/iterative) solver to achieve a balance in robustness and scalability. Domain Decomposing Linear Solver uses distributed memory in its implementation and it is not a recursive algorithm.

CHAPTER 3

METHODS AND MOTIVATION

In the previous chapter, we have presented three solution algorithms and shown their characteristics and their relation to our algorithm. This chapter will explain some methods and features that we use in our algorithm. They are domain decompostion, direct solution, shared memory and recursion.

3.1 Domain Decomposition

Domain decomposition corresponds to the methods that are flexible for the solution of linear and non-linear systems arising from the discretization of partial differential equations. Domain decomposition methods use underlying features of the partial differential equations to obtain faster solutions. They are often being used as preconditioners for linear problems and non-linear problems [14].

Their potential for efficient parallelization using data locality, ability to deal with partial differential equations even on complicated physical geometries, ability to deal with partial differential equations which have different behaviour on different parts of the domain, and superior convergence properties for iterative methods even on sequential machines are some of the motivations for the use of domain decomposition methods [15].

The main similarity of the algorithm with the one it is based on, is the method used for division of the problem into sub problems; namely domain decomposition. Domain decomposition is usually used to find preconditioners for iterative methods. For a certain linear system, any domain decomposition algorithm written in terms of functions and operators, it is possible to transform it into matrix form as a preconditioned iterative form [16]. However, in this thesis, domain decomposition is used as a part of a direct algorithm and it is independent of the problem domain. Hence, it can be referred as "algebraic domain decomposition".

Since parallel programming relies on the idea that it might be possible to have a better performance, if the solution could be achieved with using different processes at the same time on the sub problems of the same domain, it is important how we split the problem into sub problems. Because we need communication, technically it is not possible to achieve ideal speed-up. However, we can evaluate the algorithm on how much it is close to the ideal parallelism, for example on the communication volume. The problem is that the separate sub problems may need to communicate during the execution. Since the processors are getting faster and faster, the significance of the communication time is also getting more important. The advantage of domain decomposition is that we do not need to communicate between the sub problems during the execution. The partitioning method used on this algorithm allows us to have a completely independent subsystem, thus achieving minimum need for communication or data sharing. It also helps us to decrease the size of the reduced system by gathering the non-zero values on diagonal blocks. In this work, a sequential graph partitioning and fill-reducing matrix reordering algorithm, METIS [17] is used for domain decomposition.

3.2 Direct Solution

One of the features of the algorithm introduced in this thesis is that it is not used as a solver for the preconditioned system in an iterative method, but as a direct solver.

Preconditioning is a technique used for iterative algorithms in order to improve the spectrum of the initial matrix so that it is more favorable for the iterative scheme. The application of preconditioner is usually denoted by just multiplying its inverse with both sides of the equation. In practice, however, it involves solving systems where the coefficient matrix is the preconditioner at each iteration.

Iterative methods are based on the idea that we can get approximate results which are accurate enough, incrementally improved and is expected to be be computationally cheaper than the direct solvers. Coming up with an effective preconditioner, however, is often not straightforward and there are no black-box techniques to do it. That is why it is often referred as “the art of preconditioning” [18]. The scalability of direct methods are poor with problem size in terms of operation counts and memory requirements particularly for problems that arise from the discretization of partial differential equations in three space dimensions. For such cases, iterative methods are considered as the only option available [19].

Although the result of iterative methods are accurate and robust enough for many applications, they are not robust enough for many others [20]. Direct solvers are “robust”, i.e., they can solve systems that are not possible to solve with iterative solvers. There are so many direct methods each specialized on handling one or more issues such as scalability, time consumption, large memory usage, etc. [21, 22].

To solve sparse linear systems directly, there are different decompositions like Cholesky factorization, QR factorization, LU factorization and their different implementations like SuperLU [7, 8], MUMPS [23], PARDISO [24, 25, 26, 27], UMFPACK [28, 29, 30, 31], can be taken into account.

Although, the algorithm offered in this thesis is a direct algorithm, it can easily be modified to be used as a solver for the preconditioned linear system of an iterative algorithm.

3.3 Shared Memory and Recursion

Our algorithm is implemented as a multi-threaded algorithm which uses recursion on the reduced system. It is implemented using Intel Cilk Plus [4] which is an extension of programming language C and is a part of Intel Compiler. We use Cilk environment mainly because of the recursion support it has, and Intel Cilk Plus for its well documentation and support.

3.3.1 Distributed Memory versus Shared Memory

The algorithm is implemented for shared memory architectures rather than distributed memory architectures. The main difference of these two is that while distributed memory model use different processes to execute different tasks, shared memory algorithms use threads.

A process uses a virtual address space to hold the collection of program, data, stack and attributes defined in the process control block; namely process image [32]. Unlike processes, threads of a process has the same state and resources of that process, and all the threads share them since they belong to the same address space and have access to same data [32].

We now discuss the advantages and challenges of having shared memory or using threads in a parallel algorithm, specifically in our algorithm.

Distributed memory systems need a message passing method. Thus, the overheads of message passing program is usually higher than thread overheads of a shared memory system.

• The creation time of a thread is much less than a process, since each process uses it's own process image unlike the threads [32, 33].

• Since we work on large matrices, resource sharing is also advantageous. Send and receive of such big data would require more memory references and also require more storage [33].

• Scalability is also an advantage, since most of the computers are now multi-core with 2 or more computational units sharing the same main system memory. Threads may run in parallel on different cores, while a single-threaded process is able to run on only one processor [33].

• We want to use recursion in our algorithm. Another advantage is that there are shared memory programming languages which are suitable for recursive approach. While implementing a recursive code in a message passing environment, systems such as MPI [34], would be impractical.

There are also challenges that makes it difficult to implement the algorithm using shared memory.

• It is very important to divide the jobs and the work on data balanced. This challenge applies not only on shared memory systems, but also on distributed memory systems. To handle, we use a reordering and partitioning algorithm before distributing the workload of the threads.

• Another issue is data dependency. Threads working on same data concurrently means a potential danger for data corruption. There are some strategies to avoid this issue, by ensuring the synchronization. However, another issue would be that these strategies make the algorithm more difficult to code, test and debug [33]. Nonetheless, the threads of our algorithm barely needs to work on same spaces of the data, and Intel Cilk Plus [35], the programming language that we use, handles these parts of the code easily. So, data dependency is not an important problem for our current work.

• The main challenge of using shared memory for algorithm is the resources of computing environment. While using distributed memory approach, the programmer can work with all the nodes of a parallel computer, it is only possible to use the processors of one node to fully benefit from the advantages of shared memory algorithms. This issue limits our numerical experiments to use a limited number of process. This number is determined by the number of processors of a core of the computing environment we use.

To implement an algorithm with threads there are several options such as using POSIX Threads, OpenMP and Cilk.

POSIX Threads are the most known multithreading specifications referring to the POSIX standard (IEEE 1003.1c) that defines an API for thread creation and synchronization. Mostly UNIX-type systems implement POSIX Threads specification [33].

OpenMP is an API (Application Programming Interface) developed to enable portable shared memory parallel programming and to support the parallelization of applications from many disciplines. It also permits an incremental approach for parallelizing an existing code [36].

Cilk is an extension of programming language C. It is a parallel multithreaded language designed to make high-performance parallel computing easier. The programmer of a Cilk code does not have to deal with many low-level implementation details like load balancing and communication [37, 38, 39]. Cilk also allows programmer to implement a recursive parallel algorithm.

3.3.2 Recursion

Recursion can be simply described as calling the same set of routines as a part of these routines. A simple recursive function has two parts. A base case and a set of rules to reduce the system to the base case [40].

Recursive approach is used in our algorithm to execute the same processes in a similar way when proceeding. The main motivation for us to use recursion is that as the number of partitions increase, the reduced system size increases and hence the need for solving the reduced system in parallel. Since the structure of the reduce system is somehow similar to the original system, we take advantage of the partitioning of the original system and apply the same algorithm recursively on the reduce system in order to reduce the overall solution time and improve scalability.

It might be helpful to use recursion on larger matrices. The degree of diagonal dominance of the matrix could affect the performance gained by recursion. We expect that it might affect the sparsity of the reduced system matrix obtained after partitioning and DS factorization. If the initial matrix that we use is not diagonally dominant, we expect that the reduced system would be a dense matrix, so using recursion would not be beneficial. Otherwise, we can consider the reduced system as another sparse linear system and apply our algorithm recursively. For very large matrices, we expect that the reduced system would be also large enough to apply recursion efficiently.

CHAPTER 4

THE ALGORITHM

This chapter will describe not only one algorithm but all variations of the algorithm that are being used in the numerical experiments. We will also give some theoretical expectations about these variations.

Algorithm 4 The Algorithm

```
 1: procedure RDS($A, x, f, t$)          ▷ to solve $Ax = f$ with $t$ number of threads
 2:     $D + R \leftarrow A$
 3:     Indentify non-zero collumns of $R$ and store their indices in $c$
 4:     $G \leftarrow D^{-1}R$                ▷ using sparse right hand side feature of
                                           PARDISO
 5:     $g \leftarrow D^{-1}f$
 6:     $S \leftarrow I + G$
 7:     if $t \geq 4$ then
 8:         RDS($S_{(c,c)}, x_{(c)}, g_{(c)}, t/2$)
 9:     else
10:         $x_{(c)} \leftarrow S^{-1}_{(c,c)} g_{(c)}$
11:     end if
12:     a. $x \leftarrow g - G_{(:,c)} x_{(c)}$
        b. $x \leftarrow D^{-1}(f - R\hat{x})$   ▷ where $\hat{x}$ is the full sized vector filled with
                                           only the values corresponding indice $c$
13: end procedure
```

Given a general sparse linear system of equations:

$Ax = f$ is assumed to be the sparse linear system after METIS reordering and we partition the matrix and the right hand side based on the information provided by METIS [17] for gathering the non-zero values in the diagonal blocks.

In Figure 4.1, a sparse linear system of equations, $Ax = f$, is given. Dark green cells, white cells and grey cells represent non-zero blocks, zero blocks and unknown blocks

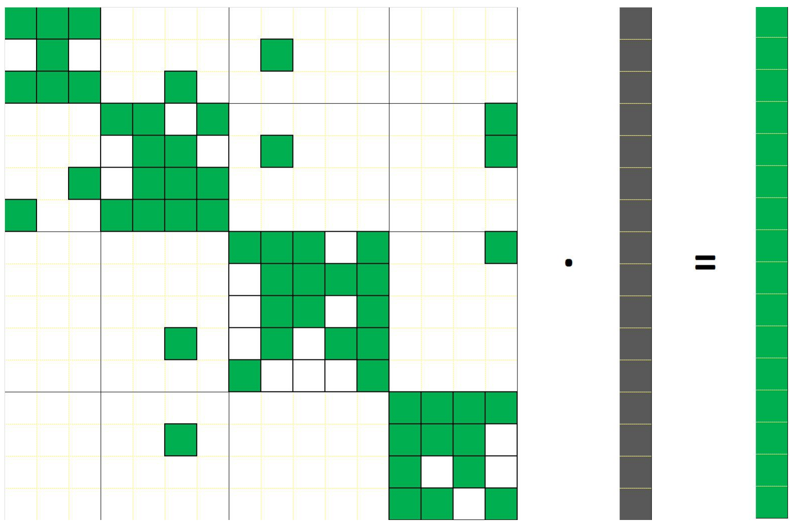
Figure 4.1: A linear system of equations $Ax = f$

consecutively.

The factorization is considered to be the preprocessing stage of the algorithm since it can be computed only once if linear systems with the same sparsity structure of the coefficient matrix needs to be solved. Here, like other well-known algorithms, we are factorizing the initial sparse matrix A to use this factors in the solution phase. Substeps 2,3 and 4 (of algorithm 4) are all considered to be preprocessing since they do not use the right side of the linear system to be solved, namely f.

Solution of linear systems involving the diagonal blocks in the algorithm are handled using a well known parallel direct sparse linear system solver, PARDISO [24, 25, 26, 27]. Other direct or iterative solvers, however, can easily replace PARDISO provided they have the capability to compute partial solution in case of sparse right hand side. To improve data locality; summation, subtraction and multiplications are handled using BLAS and Sparse BLAS routines of Intel MKL [41, 42].

The algorithm is almost ideally parallel. There is no communication before the solution of the reduced system. Since our approach is recursive, the time consumed solving this part is also expected to be reduced.

In general, all sparse linear solvers find ways to handle the non-zero values of the initial matrix without losing time with the zeros. To do this and reduce the fill-in, domain decomposition is used in our algorithm.

$$D + R \leftarrow A \tag{4.1}$$

Here, D consists of the block diagonals of the matrix A. The number of block diagonals is determined by the number of threads that will be used. Each block diagonal will be

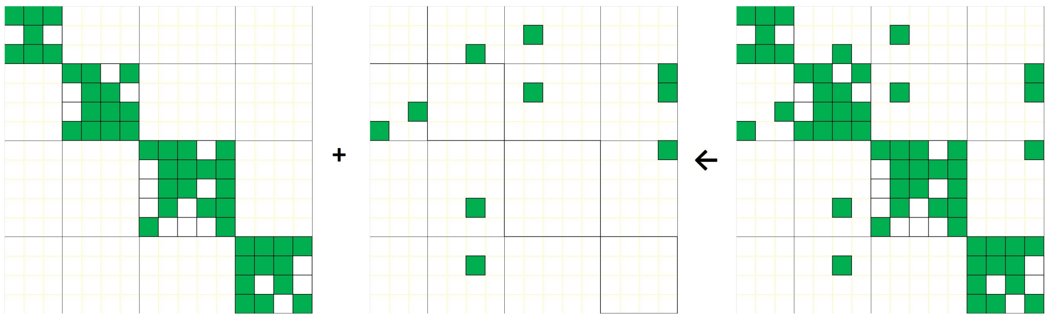

Figure 4.2: $D + R \leftarrow A$

handled by different threads on preprocessing stage.

Matrix R consists of the remaining elements, i.e., the values of A which are not inside the diagonal blocks. We can compute R by subtracting D from A, i.e., $R = A - D$. The non-zero columns of R determine the size of the reduced system.

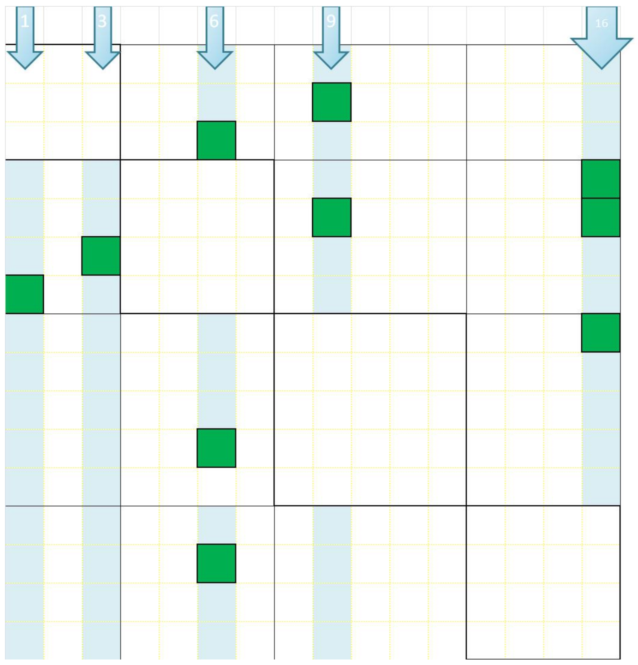

Figure 4.3: Matrix R with non-zero columns marked

Here in our example, we just need to process only 5 columns out of 16 of the matrix R. From now on, we are operating only on the selected indices of the matrix for all the steps but the last step.

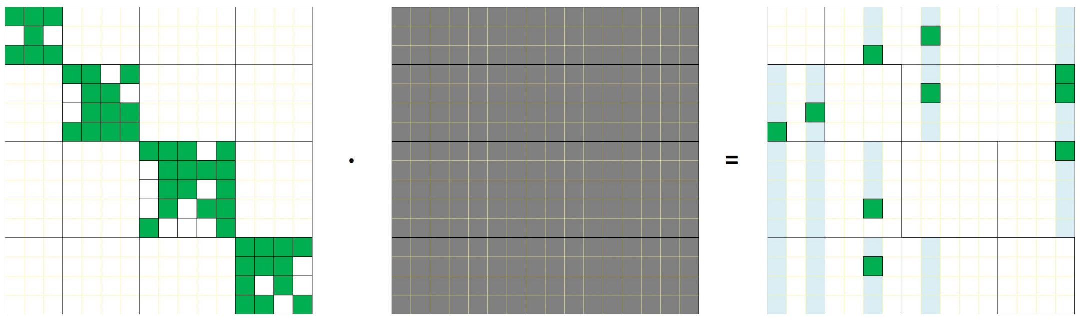
Figure 4.4: Solve $DG = R$ for only non-zero collumns shown with light blue

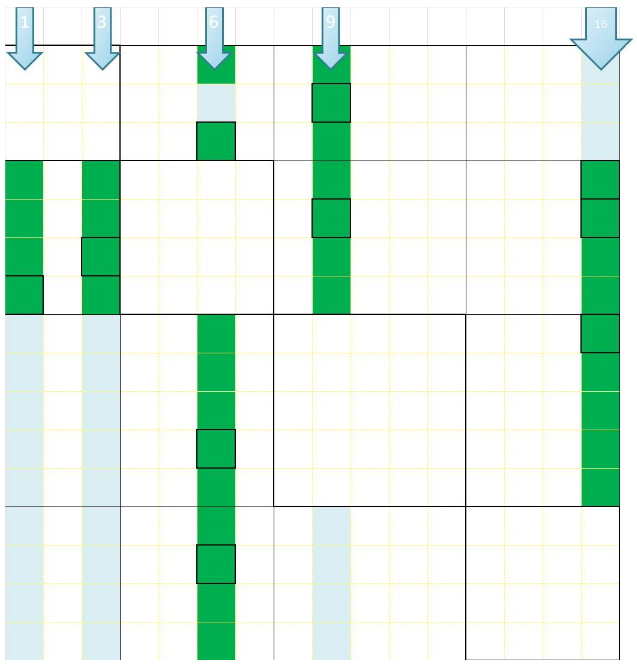

Figure 4.5: Matrix G with non-zero columns marked

$$G \leftarrow D^{-1}R \tag{4.2}$$

$G \leftarrow D^{-1}R$ is applied in parallel on each processor. The matrix is partitioned row-wise and each process executes the computation on its own part of D and R.

We now have a structure with non-zero values only on the columns where R has non-zero values. Since we are going to apply the same operation on the right side of the initial equation, the solution remains unchanged, i.e., $((D^{-1}A)x = D^{-1}f)$, where $Ax = f$. After this operation, a smaller and independent system, the reduced system, can be extracted and solved. Note that it was not possible to find and independent and small system before this step.

The most time consuming part of the algorithm is this part, where we compute the matrix G. We expect that R is also a sparse matrix. Since we have this sparse matrix on the right side and we do not need every entry in the solution, some savings could be made by using the sparse right hand side feature of PARDISO.

The main advantage is that we do not have to calculate the values in the solution vector that are in the same row with the zero values of matrix R resulting in reduction of the amount of computation, and hence a reduction in total time. However, this could also mean some information loss since some entries of G matrix will not be computed. This means a trade-off between speed and robustness. If we use sparse right hand side feature of PARDISO, the algorithm is faster but less robust.

The magnitude of values that we do not compute mainly depends on the degree of block diagonal dominance of matrix D. Thus, if the initial matrix has very large values on it's diagonal blocks, the values we lose because of using sparse right hand side feature of PARDISO becomes less important since they are practically zero. So, the robustness of the algorithm depends on the block diagonal dominance after the permutation, i.e., the dominance of the values on the block diagonals with respect to the other values on the matrix.

Nonetheless, not all the matrices are block diagonal dominant matrices after METIS reordering. So, we may need a method that still benefits from the sparse right hand side feature of PARDISO and maintain the robustness regardless of block diagonal dominance. To do this, we need to find the solution of the system without using the values in G which are not calculated due to the usage of sparse right hand side feature of PARDISO. Such method is given on 12.b (Algorithm 4).

Note that we calculated some values that corresponds to the zero values of the right hand side vector. They are calculated because they are needed for step 12 (of Algorithm 4), to find the reduced system correctly. However, for systems that have very large block diagonal dominance, skipping to calculate these values may also lead to a speed-up without a significant loss of precision.

In Figure 4.6, red cells denote the values that are not calculated, light greens are the cells that are calculated because we need them in step 12 of Algorithm 4.

Since we have now all the necessary information for the left hand side of the reduced system, we can now move on to the operations involving the right side. Until now, we did not use any value from the right hand side of the system, and we also did not need any communication.

$$g \leftarrow D^{-1} f \tag{4.3}$$

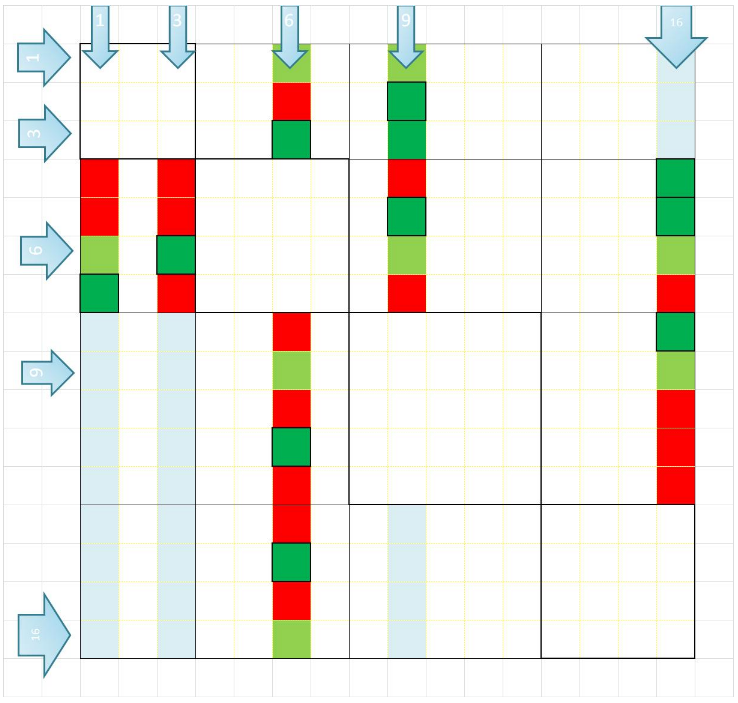

Figure 4.6: Matrix G with red cells that may cause loss of precision

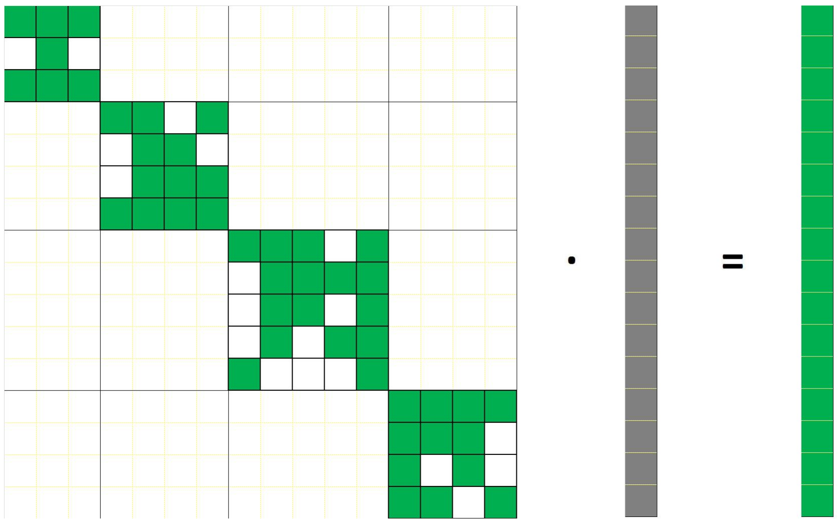

Figure 4.7: Solve $Dg = f$

To solve $Dg = f$, We apply D^{-1} to the right hand side of the initial linear system. Note that, again we are solving multiple independent linear systems where the coefficient matrices are the blocks of the matrix D and we do not compute D^{-1} explicitly. Since in general the right hand side vector is assumed to be dense, we do not use the sparse right hand side feature of PARDISO in this step. This operation does not require too much time because it is only forward and backward sweeps of already computed LU factorization and is completely parallel.

$$S \leftarrow I + G \tag{4.4}$$

The operation $I + G$ actually gives the result of $D^{-1}A$, since $A = D + R$ and $D^{-1}A = D^{-1}(D + R)$, which is also equal to $I + G$.

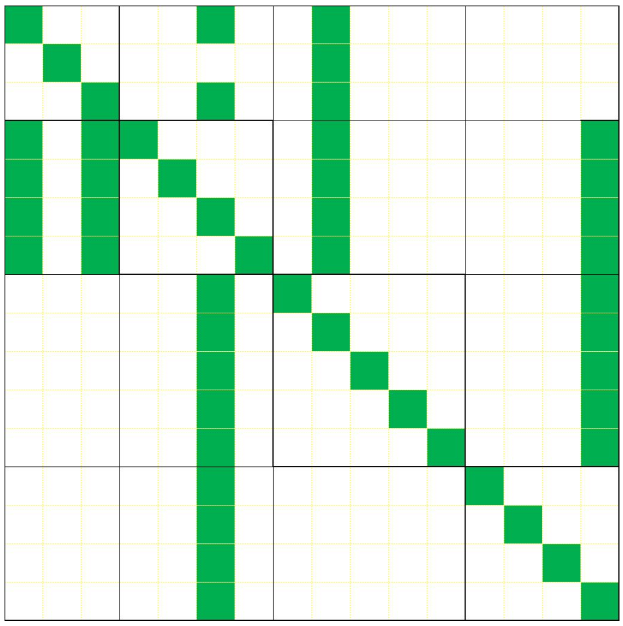

Figure 4.8: Matrix S

After we drop the indices that we will not use for the reduced system, $S_{(c,c)}$ is shown on Figure 4.9. Dark green cells denote the values that are absolutely necessary for any case and light green cells contain the values that might be dropped in case the initial system would be block diagonally dominant.

Even if we use the sparse right hand side feature of PARDISO, there is no information loss on $S_{(c,c)}$ and hence it is exact. There is no loss of information, because while calculating G, we calculate all the needed values for the reduced system, namely the values that are on the rows that are corresponding to the indice c.

However, we may not have to compute the entries of the matrix that are expected to be relatively small if the matrix is block diagonally dominant. In this small example these are highlighted as light green. This means some information is lost, but both

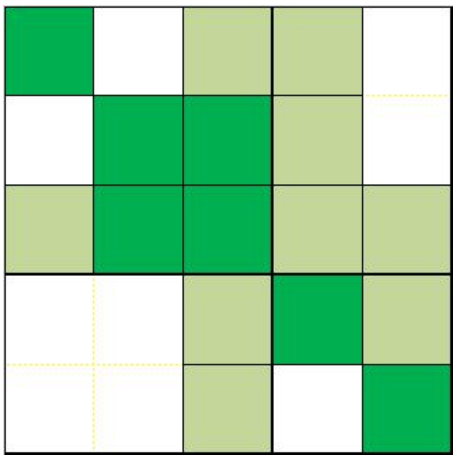

Figure 4.9: Matrix $S_{(c,c)}$ with some values that might be dropped shown with light green

computing the reduced system and solving the reduced system will be faster. We can also accelerate the algorithm that way with a trade-off in terms of robustness. Again, the accuracy loss would depend on the block diagonal dominance of the initial system. It might be possible to implement an extra preprocessing phase that first increases the block diagonal dominance with an additional preprocessing and apply the algorithm on this system. Nevertheless, this is out of the scope of our current work.

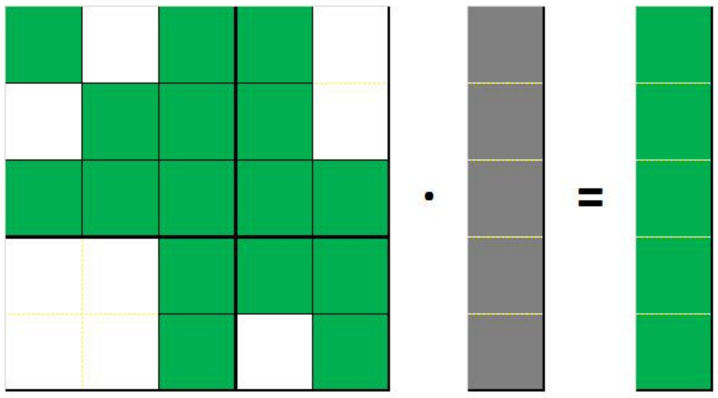

Figure 4.10: Solve $S_{(c,c)}x_{(c)} = g_{(c)}$

$$x_{(c)} = S_{(c,c)}^{-1} g_{(c)} \tag{4.5}$$

In this example, the reduced system is considerably small and hence, we do not need to use recursion. However, if we consider filled cells as an abstraction, for example as block sparse matrices consisting of matrix entries at least one having a non-zero value, the reduced system could be still sparse and large enough for recursion.

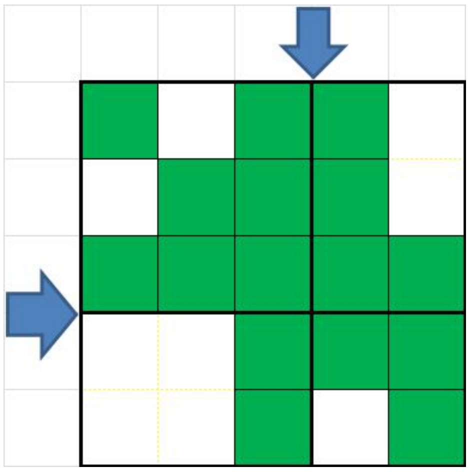

Figure 4.11: Matrix $S_{(c,c)}$ with cut points for recursion shown

Since calling METIS on every recursive step would be expensive, we rather choose our partitioning points for the blocks referring to the last cut-off. In this example the first 3 blocks come from the thread 1 and 2, and the last 2 from the thread 3 and 4. We halved the number of threads, thus we cut from where thread 2 finishes.

The sparsity of the reduced system is observed to be related to the degree of diagonal dominance of the initial Matrix A. If the initial Matrix is diagonally dominant, the values resulted from the equation $G = D^{-1}R$ would be nearly zero, so the reduced matrix would be sparse.

For the base case of recursion, we used PARDISO in our algorithm to solve the reduced system. However, if the reduced system is dense, it would be more suitable to use a dense linear system solver such as the ones in LAPACK [43] or ScaLAPACK [44], since dense solvers are more efficient for dense matrices.

$$x \leftarrow g - G_{(:,c)}x_{(c)} \tag{4.6}$$

Unless we use sparse right hand side feature of PARDISO, this operation works correctly and gives the exact result, because the solution of the reduced system is also exact. The proof of the formula is given by Proof 1.

Proof 1

```
1: $Gx = G_{(:,c)}x_{(c)}$                ▷ as $G$ has non-zero values only on their indices $c$.
2: $DGx = DG_{(:,c)}x_{(c)}$
3: $Rx = DG_{(:,c)}x_{(c)}$               ▷ since $G = D^{-1}R$, i.e., $DG = R$.
4: $f = Dx + DG_{(:,c)}x_{(c)}$           ▷ since $f = Ax$, and$A = D + R$, thus $f = (D + R)c = Dx + Rx$.
5: $D^{-1}f = x + G_{(:,c)}x_{(c)}$
6: $g = x + G_{(:,c)}x_{(c)}$,            ▷ since $g = D^{-1}f$.
7: $x = g - G_{(:,c)}x_{(c)}$
```

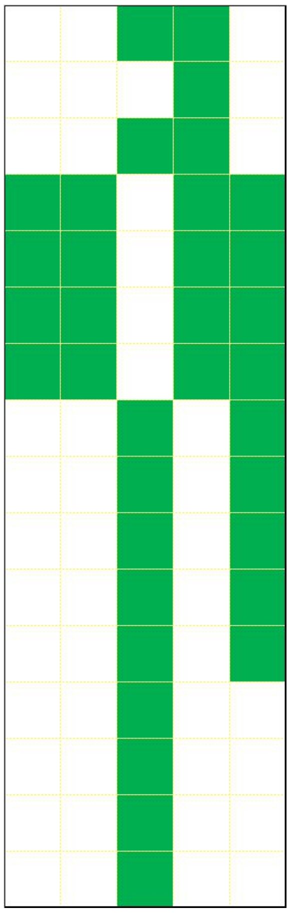

Figure 4.12: Matrix $G_{(:,c)}$

$$x \leftarrow D^{-1}(f - R\hat{x}) \tag{4.7}$$

Because the speed is important for us, we used sparse right hand side feature of PARDISO almost on all of the numerical experiments. $G = D^{-1}R$ is approximately calculated if we use this feature. There is information loss on matrix G. So, the result is also approximate. If the matrix is block diagonally dominant, the unused values are small, so the result is more robust. Otherwise, we can use another approach which is also exact at a cost of another parallel linear solve using the block diagonal matrix D as the coefficient matrix. We note that the extra cost here is not too much since the factors for each diagonal block of D has been already computed in the first stage of algorithm.

We can calculate x by 12.b, where $\hat{x}$ is a vector in which only values corresponding to the indice c is filled by the values of $x_{(c)}$. Since we do not use the matrix G on this computation, the sparse right hand side feature of PARDISO does not affect the accuracy of our result. The proof of this method is given by Proof 2.

Proof 2

```
1: $Rx = R\hat{x}$                      ▷ since $R$ has non-zero values only on the
                                          columns that corresponds to the indices
                                          shown by $c$.
2: $Ax = f$
3: $Dx + Rx = f$                        ▷ since $A = D + R$.
4: $Dx + R\hat{x} = f$                  ▷ since $Rx = R\hat{x}$.
5: $Dx = f - R\hat{x}$
6: $x = D^{-1}(f - R\hat{x})$
```

CHAPTER 5

NUMERICAL EXPERIMENTS

In the previous chapters, we have explained the methods that we used and our algorithm. This chapter will show the programming and computing environments we worked on and give the experimental results of our work.

5.1 Programming Environment

In this section, we will give the information about the libraries we used in our algorithm. We will explain their features and how we used them.

METIS is a sequential graph partitioning and fill-reducing matrix reordering algorithm [17]. “The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes” in Karypis Lab [45]. It is used in our work to gather the non zero values on matrix near the main diagonal by setting it to minimize the communication volume. METIS finds a symmetric permutation of matrices and works on undirected graphs. For non-symmetric matrices, we used a symmetric matrix that matches to the structure of our non-symmetric matrix, i.e., dividing all the values on $(|A|^T + |A|)$ by 2. Version 5.1.0 of METIS is used in our work.

Some of the matrices that are used in our experiments has zero values on their main diagonals. Since having even one zero value in the main diagonal means that our matrix is singular (or non-invertable), we apply non-symmetric permutation. HSL MC64 is a collection of Fortran codes to find a column permutation vector to ensure that the matrix will have only non-zero entries on its main diagonal [46]. The permutation vector created by HSL MC64 is used on some of our test matrices.

Intel Cilk Plus is an extension of C and C++ languages [35]. In our work it is used to ensure efficient multi-threading with recursion.

Intel MKL (Math Kernel Library) is a library of math routines such as BLAS, LAPACK, ScaLAPACK and sparse linear system solvers [41]. In our work, all the basic matrix-vector and matrix-matrix operations are handled using BLAS and Sparse BLAS routines of Intel MKL version 10.3 [42].

PARDISO [24, 25, 26, 27] is a well known, fast and robust parallel direct solver for sparse linear systems of equations. It works both on shared and distributed memory architectures. It is used both for testing our results and as a part of our algorithm. All of the solutions inside of our algorithm are handled by PARDISO. Version 4.1.2 of PARDISO is used in our work. For factorization phase, it applies fill-in METIS reordering by default. The only non-default parameter that we use is the one that controls the sparse right hand side feature of PARDISO. When the right hand side is also a sparse matrix (see line 4 of Algorithm 4), we use this option to avoid unnecessary calculations and save time.

5.2 Computing Environment

All the numerical experiments were executed on NAR. NAR is the High Performance Computing Facility of Middle East Technical University Department of Computer Engineering. "It aims to serve research studies and courses of METU, involving parallel algorithms and solutions" [47].

A single node of NAR contains 2 x Intel Xeon E5430 Quad-Core CPU (2.66 GHz, 12 MP L2 Cache, 1333 MHz FSB) and 16 GB Memory. Nar uses an open source Linux distribution developed by Fermi National Accelerator Laboratory (Fermilab) and European Organization for Nuclear Research (CERN), Scientific Linux v5.2 64bit, as its operating system [47].

5.3 Experiments and Results

In this section, we will discuss the results of the tests. For each test matrix we use two figures and a table. The number of threads used for each test starts from 1 and continues with the powers of 2. Because of the limitations of our computing environment, we used at most 8 threads.

For each different thread number there are 5 test cases:

1) A well-known parallel direct solver for sparse linear sytem of eqations, PARDISO.

2) Non-recursive DS factorization using the sparse right hand side feature of PARDISO in its computations (DS-nr-sp).

3) Non-recursive DS factorization without using the sparse right hand side feature of PARDISO (DS-nr-ns).

4) Recursive DS factorization using the sparse right hand side feature of PARDISO (DS-re-sp).

5) Recursive DS factorization without using the sparse right hand side feature of PARDISO (DS-re-ns).

The matrices used for testing are retrieved from University of Florida Sparse Matrix collection [4]. The properties of matrices are given on Table 5.1. For indefinite matrices, the degree of diagonal dominances given are calculated after non-symmetric permutation.

Table 5.1: Properties of Test Matrices

Matrix Name	**Number of Rows and Columns**	**Number of Non-zeros**	**Degree of Diagonal Dominance**	**Problem Type**
ASIC_320k	321,821	1,931,828	0.000383	circuit simulation
ASIC_680ks	682,712	1,693,767	0.000001	circuit simulation
crashbasis	160,000	1,750,416	2.701016	optimization
ecology2	999,999	4,995,991	1.000000	2D/3D
Freescale1	3,428,755	17,052,626	0.491856	circuit simulation
hvdc2	189,860	1,339,638	0.000036	power network
Kaufhold	160,000	1,750,416	0.000003	counter-example
Kuu	7,102	173,651	0.255813	structural
Lin	256,000	1,766,400	1.931723	structural
majorbasis	160,000	1,750,416	0.032386	optimization
Pd	8,081	13,036	0.000015	counter-example
powersim	15,838	67,562	0.001767	power network
Raj1	263,743	1,300,261	0.000203	circuit simulation
rajat21	411,676	1,876,011	0.000002	circuit simulation
scircuit	170998	958936	0.000037	circuit simulation
stomach	213,360	3,021,648	0.166783	2D/3D
tomography	500	28,726	0.000551	computer graphics /vision
torso3	259,156	4,429,042	0.098827	2D/3D
transient	178866	961368	0.000060	circuit simulation
xenon2	157464	3866688	0.082090	materials

The results are evaluated in terms of robustness and speed. Thus, for every result discussed in this section, there will be two figures to show time and speed-up, and a table to show the relative error.

It can be seen from Table 5.1 that, out of eight different domains that we use, most of the tests held on circuit simulation problem. More circuit simulation problems are included in the tests because they provide variety of the degree of diagonal dominance, the number of rows and columns, the number of non-zero entries and the sparsity of structures.

The first matrix that we are going to look is ecology2 from 2D/3D problem domain. Its degree of diagonal dominance is 1, which means ecology2 is diagonally dominant. The structure of ecology2 is symmetric. However, our code is implemented for non-symmetric coefficient matrices. Thus, instead of taking advantage of symmetry, we changed its structure to the non-symmetric form and computed result as if it is non-symmetric.

The time values and the relative residuals are very close to PARDISO for ecology2 (Figure 5.1, Figure 5.2, Table 5.1). Nonetheless, 3 of our 4 approaches are faster for the 8-threaded case. The performance improvement of using sparse right hand side feature is obvious in this example. Moreover, the speed-up is almost ideal if sparse right hand side is used with recursion. This is mainly because of the degree of diagonal dominance of ecology2.

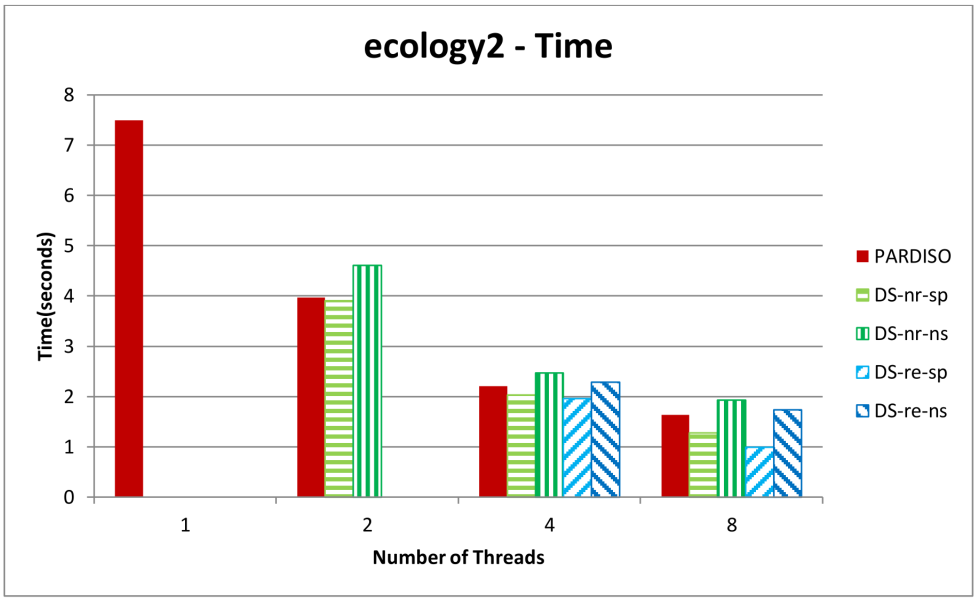

Figure 5.1: Matrix ecology2: Time versus Number of Threads

Table 5.2: Relative Residuals on Matrix ecology2

Threads	PARDISO	DS-nr-sp	DS-nr-ns	DS-re-sp	DS-re-ns
1	1.32e-16	-	-	-	-
2	1.31e-16	1.32e-16	1.37e-16	-	-
4	1.31e-16	1.31e-16	1.46e-16	2.04e-16	1.54e-16
8	1.29e-16	1.29e-16	1.45e-16	1.71e-16	1.54e-16

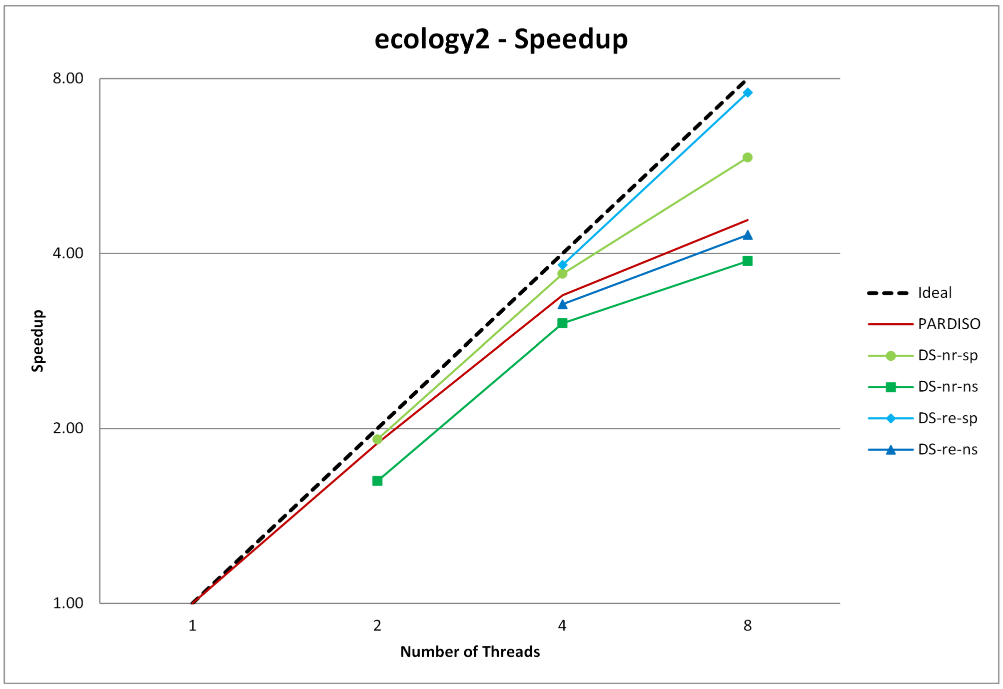

Figure 5.2: Matrix ecology2: Speedup versus Number of Threads

Freescale1 is from circuit simulation problem domain and has a degree of diagonal dominance 0.49. Thus, it is not diagonally dominant but has a reasonable degree. Freescale1 is a non-symmetric matrix. So, there is no need to change the structure before tests.

Although the speed up for tests on Freescale1 are better than or equal to PARDISO for sparse right hand side cases, unlike the tests on ecology2, our algorithm is slightly slower than PARDISO if sparse hand side feature is not used (Figure 5.3, 5.4). This might be caused by the distribution of non-zero values which are out of block diagonal blocks. Because if the non-zero values are on disjoint columns we have to deal with more columns in our factorization. On the other hand, using sparse right hand side feature of PARDISO we counter-work some of the negative effects of working on more sparse columns by omitting some unnecessary values.

Another important difference to be expressed is that our algorithm is more robust than PARDISO for this case (Table 5.3). This might be the result of the structure of diagonal blocks of matrix Freescale1.

Kaufhold is a matrix from domain of counter-example problems and has a very low degree of diagonal dominance, $3 \cdot 10^{-6}$. It can also be considered as a counter-example for our algorithm, since it does not give good results compared to PARDISO, in terms of time (Figure 5.5, 5.6). For all the test on Kaufhold, our algorithm is slower than

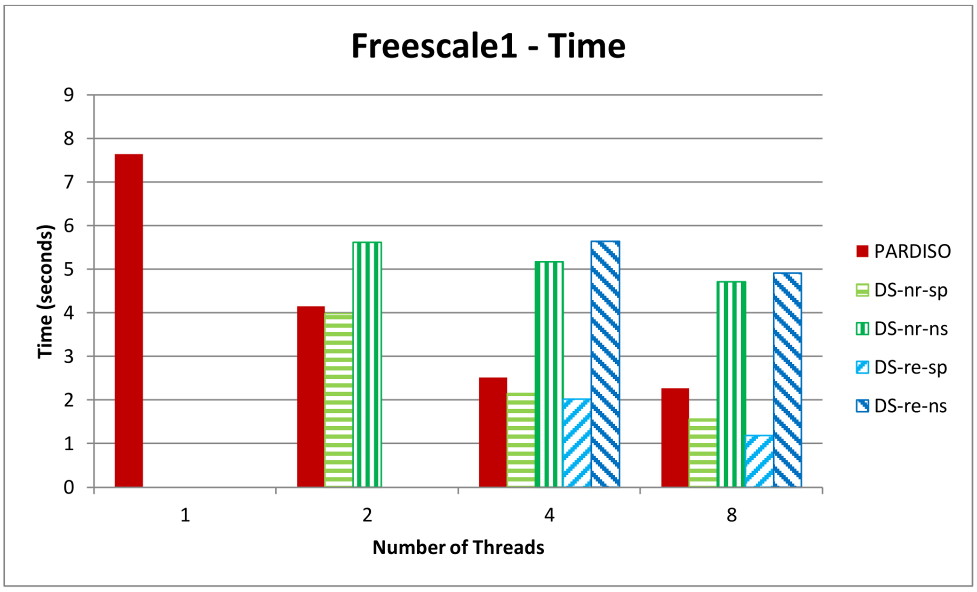

Figure 5.3: Matrix Freescale1: Time versus Number of Threads

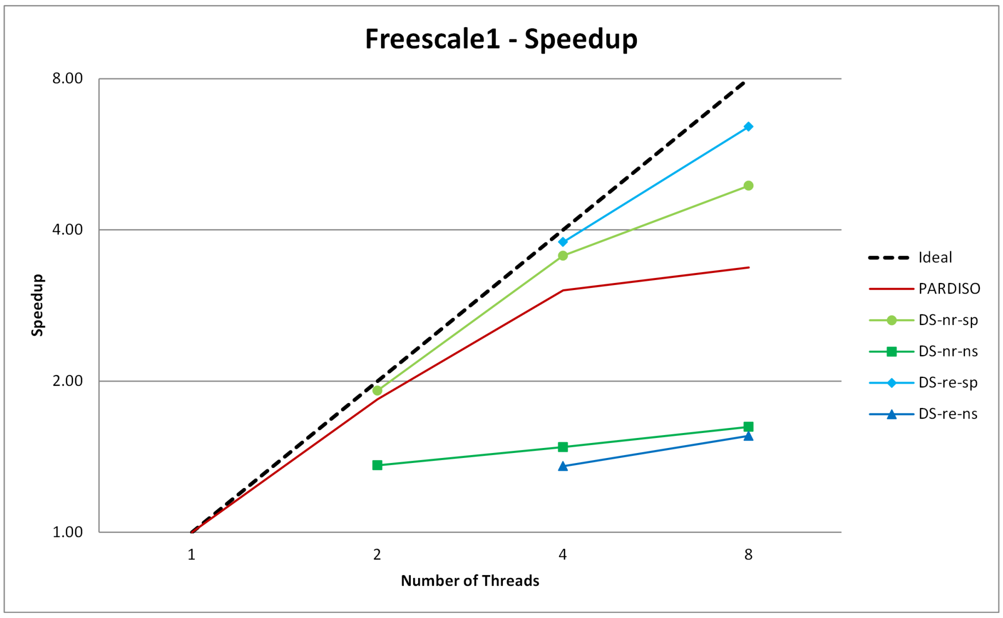

Figure 5.4: Matrix Freescale1: Speedup versus Number of Threads

PARDISO. Recursion is also slow for this example. It is an expected result. Since our algorithm is slower than PARDISO for this case, using it for the reduced system instead of PARDISO should also slow down the algorithm. Low degree of diagonal dominance of matrix Kaufhold is also a cause for low performance of recursion.

Table 5.3: Relative Residuals on Matrix Freescale1

Threads	**PARDISO**	**DS-nr-sp**	**DS-nr-ns**	**DS-re-sp**	**DS-re-ns**
1	1.79e-10	-	-	-	-
2	1.79e-10	4.47e-15	2.11e-15	-	-
4	1.79e-10	4.29e-15	9.08e-16	7.11e-15	6.71e-16
8	1.85e-10	8.64e-16	8.61e-16	1.03e-15	1.94e-16

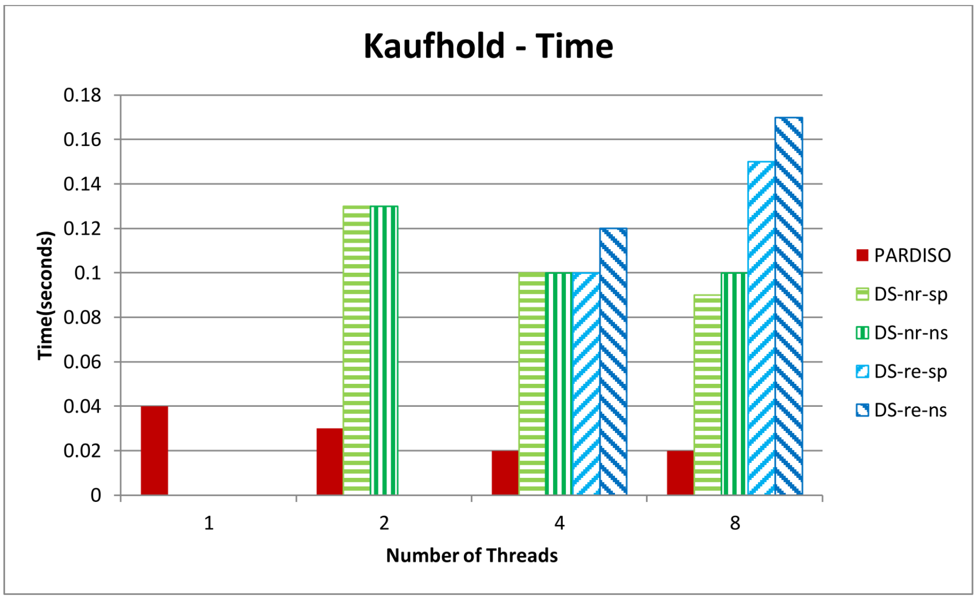

Figure 5.5: Matrix Kaufhold: Time versus Number of Threads

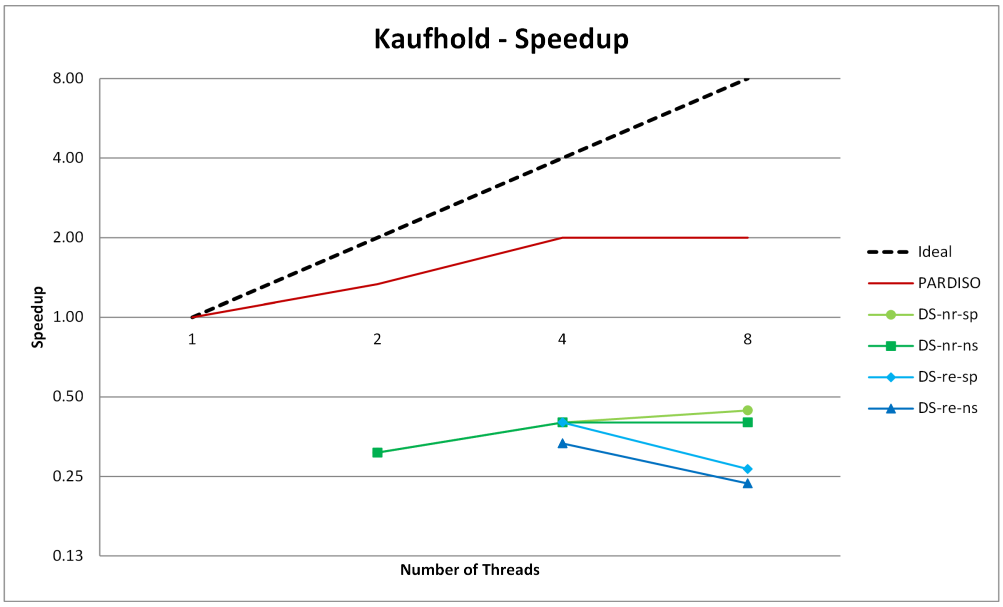

Figure 5.6: Matrix Kaufhold: Speedup versus Number of Threads

Table 5.4: Relative Residuals on Matrix Kaufhold

Threads	PARDISO	DS-nr-sp	DS-nr-ns	DS-re-sp	DS-re-ns
1	9.72e-14	-	-	-	-
2	9.72e-15	1.26e-16	1.25e-16	-	-
4	9.72e-15	1.23e-16	1.23e-16	3.14e-04	1.93e-16
8	9.72e-15	1.20e-16	1.25e-16	6.51e-16	1.37e-16

Matrix Lin is a good example where using the sparse right hand side feature of PARDISO is the main difference. It is from the domain of structural problems and has a good degree of diagonal dominance, 1.93. Lin is also a symmetric matrix. Thus, the same operation on ecology2 is also applied on Lin.

If we use the sparse right hand side feature, our algorithm works nearly as fast as PARDISO, and even faster on 8 threads (Figure 5.7, 5.8). Recursion also works very well. Nevertheless, it does give very bad results for the other cases. Unless we use sparse right hand side, our algorithm gives results even worse than single-threaded PARDISO. The distribution of non-zero values might be the cause of having bad results for also this case.

Matrix rajat21 is from circuit simulation problem domain. The difference of it is that its degree of diagonal dominance before any permutations is zero. In other words, it is a singular (or non-invertible) matrix. To handle this issue, a non-symmetric permutation (by HSL MC64) before execution is applied on these matrices. After the permutation,

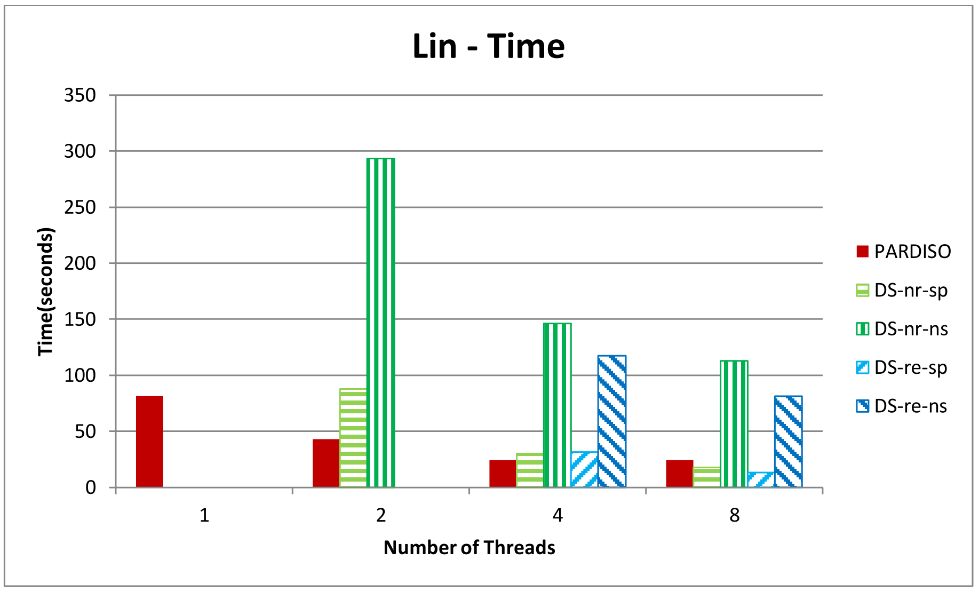

Figure 5.7: Matrix Lin: Time versus Number of Threads

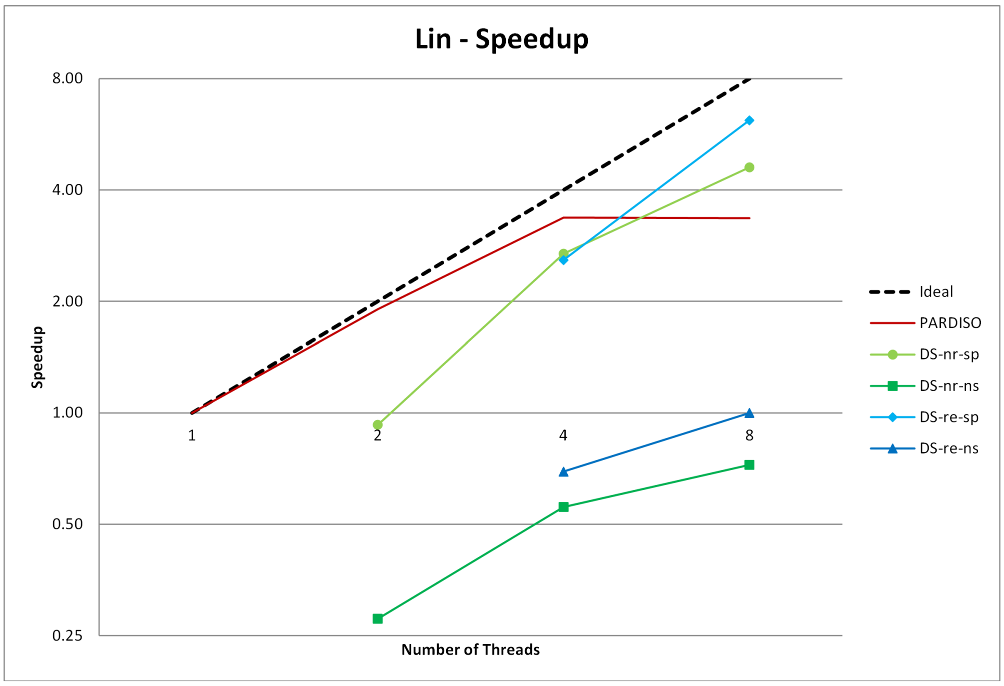

Figure 5.8: Matrix Lin: Speedup versus Number of Threads

the degree of diagonal dominance of matrix rajat21 is $2 \cdot 10^{-6}$. Therefore, it expected for the recursion to work less efficient on rajat21. When we use the sparse right hand side feature of PARDISO, our algorithm works fast even on the 8 threaded case where PARDISO gets slower (Figure 5.9, 5.10).

Table 5.5: Relative Residuals on Matrix Lin

Threads	PARDISO	DS-nr-sp	DS-nr-ns	DS-re-sp	DS-re-ns
1	9.95e-10	-	-	-	-
2	8.81e-16	5.63e-16	1.88e-16	-	-
4	8.07e-16	8.72e-16	6.99e-16	3.70e-13	1.18e-12
8	7.16e-16	1.36e-15	2.14e-16	6.13e-13	8.65e-14

The relative residual of PARDISO for rajat21 is not well, compared to the test on other matrices. Since our algorithm uses PARDISO as an inner solver, it also applies to our solvers. However, our results are still more robust than the results of PARDISO for rajat21 (Table 5.6).

The last matrices to be shown in this chapter are torso3 and xenon2 from 2D/3D and materials problem domains respectively. They differ on both the problem domains and solution times but they have very close diagonal dominances. torso3 has 0.10, where xenon2 has 0.08.

The behaviours of tests in terms of speed on both matrices are almost the same. They are at least as fast as PARDISO for non-recursive cases and they are slower in case of usage of recursion.

The relative residuals on torso3 are better than on xenon2, but the difference between robustness of our algorithm and PARDISO is very low on both cases.

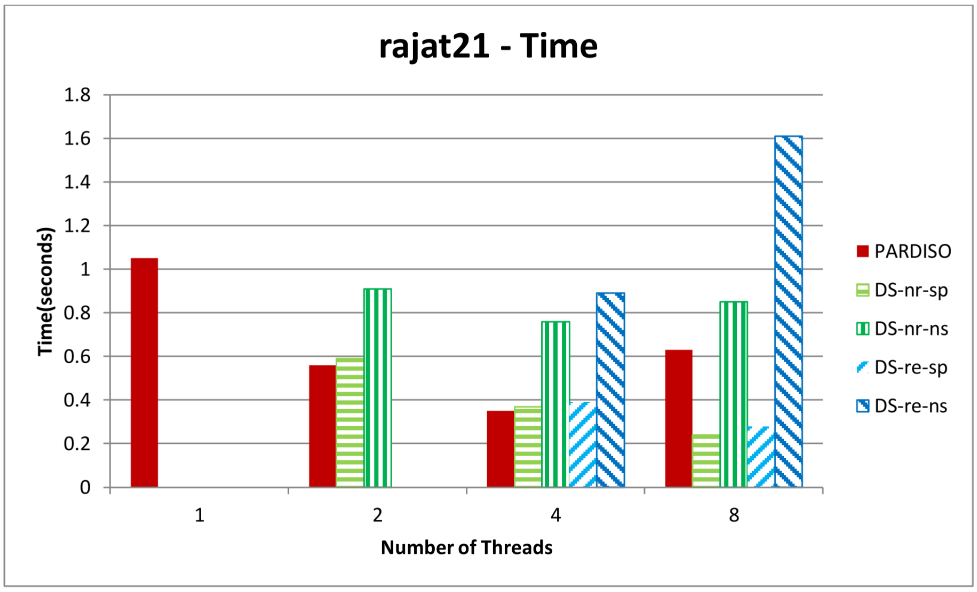

Figure 5.9: Matrix rajat21: Time versus Number of Threads

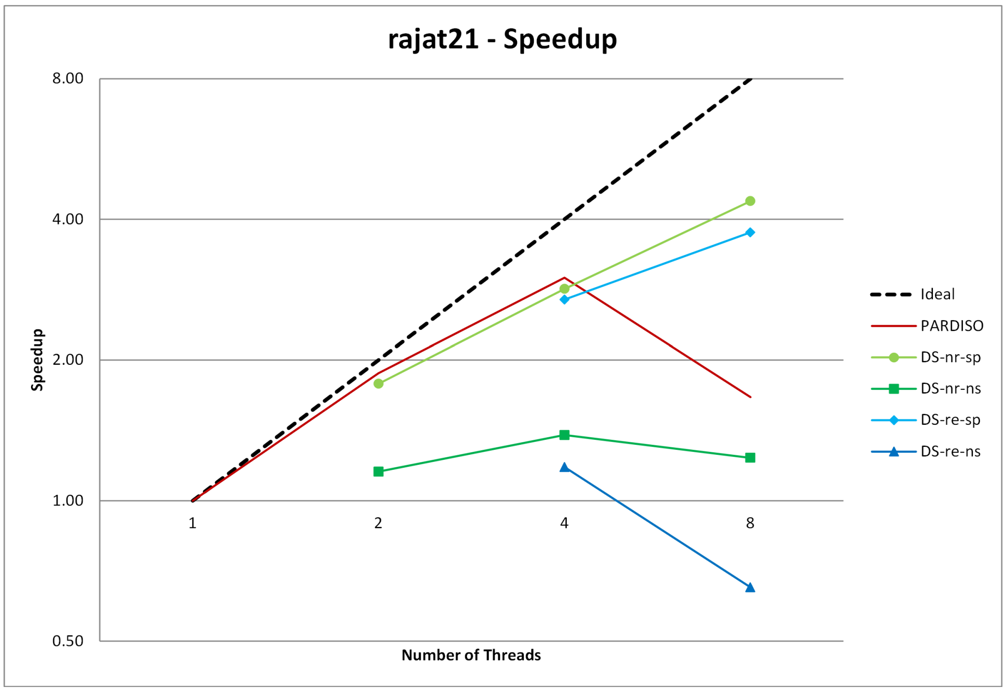

Figure 5.10: Matrix rajat21: Speedup versus Number of Threads

Table 5.6: Relative Residuals on Matrix rajat21

Threads	**PARDISO**	**DS-nr-sp**	**DS-nr-ns**	**DS-re-sp**	**DS-re-ns**
1	2.54e-05	-	-	-	-
2	7.97e-06	5.63e-16	2.51e-07	-	-
4	8.38e-06	2.65e-07	1.11e-07	3.18e-04	1.18e-07
8	1.16e-05	8.04e-08	5.68e-08	9.96e-05	5.81e-08

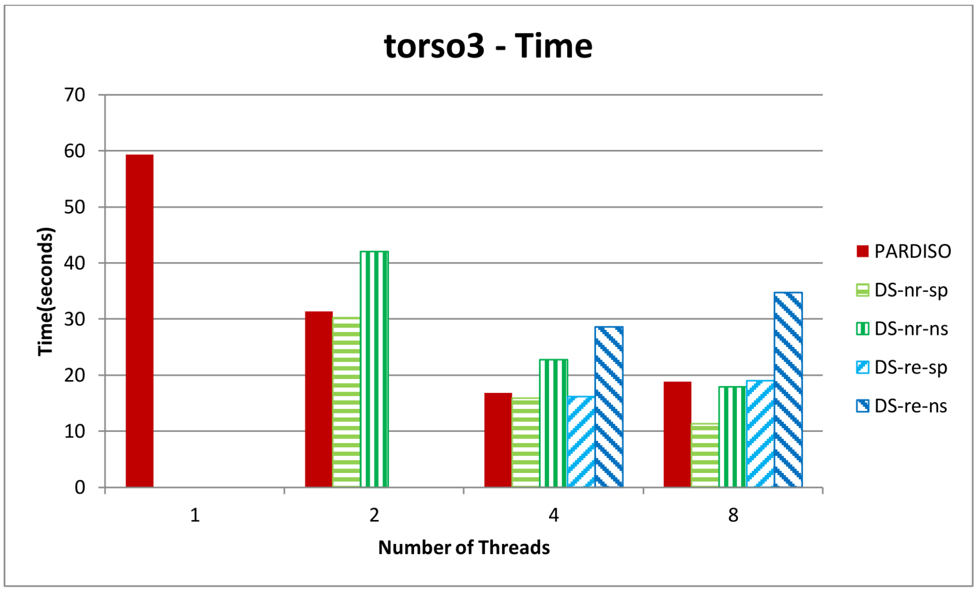

Figure 5.11: Matrix torso3: Time versus Number of Threads

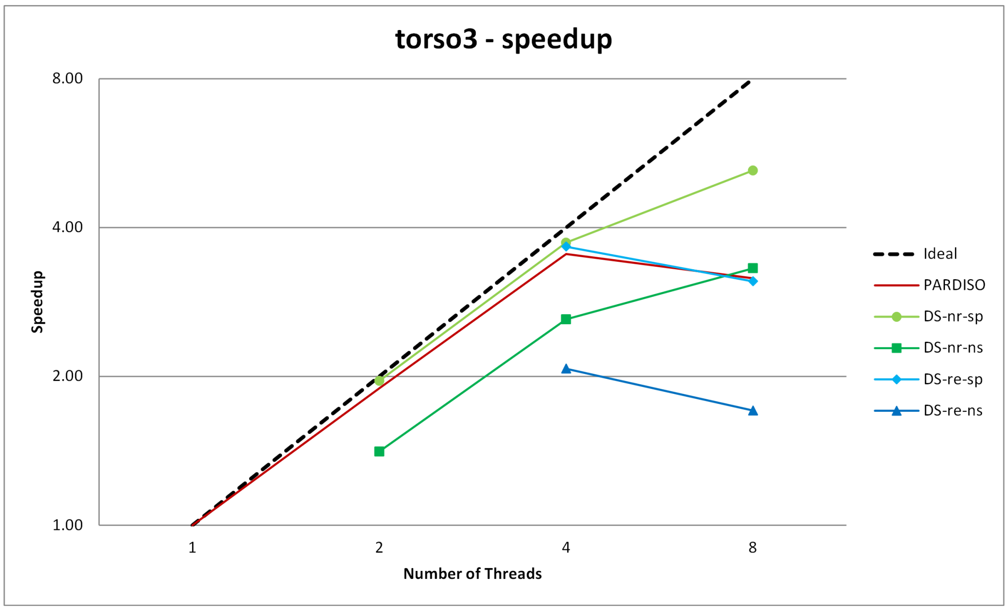

Figure 5.12: Matrix torso3: Speedup versus Number of Threads

Table 5.7: Relative Residuals on Matrix torso3

Threads	PARDISO	DS-nr-sp	DS-nr-ns	DS-re-sp	DS-re-ns
1	1.03e-15	-	-	-	-
2	1.02e-15	4.80e-15	4.24e-15	-	-
4	1.01e-15	1.74e-15	1.69e-15	2.11e-15	1.42e-15
8	9.93e-16	6.82e-16	7.83e-16	8.59e-16	4.15e-16

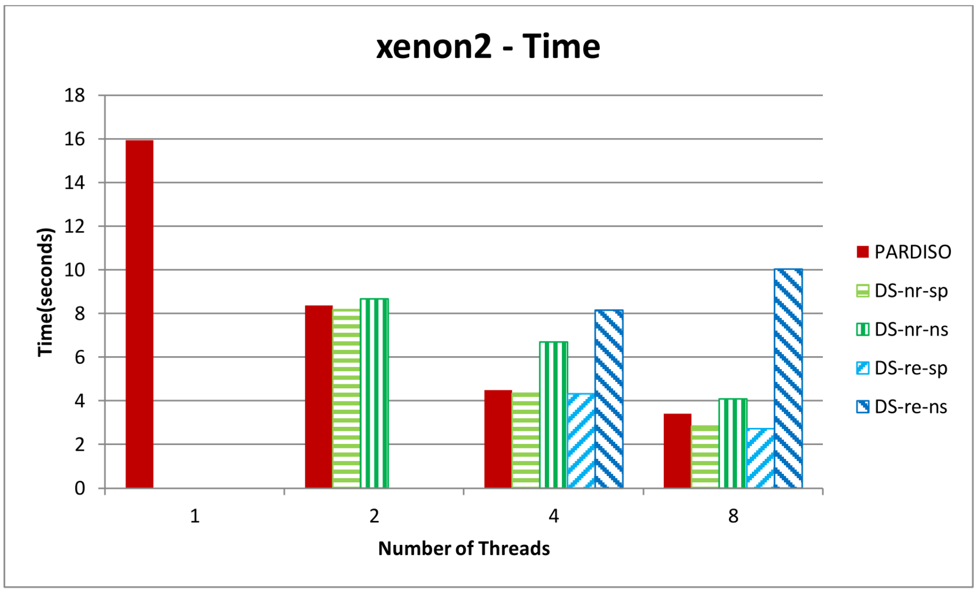

Figure 5.13: Matrix xenon2: Time versus Number of Threads

Table 5.8: Relative Residuals on Matrix xenon2

Threads	PARDISO	DS-nr-sp	DS-nr-ns	DS-re-sp	DS-re-ns
1	4.38e-12	-	-	-	-
2	4.39e-12	9.41e-12	1.12e-13	-	-
4	4.37e-12	4.33e-12	1.12e-13	2.21e-11	6.71e-12
8	4.35e-12	1.74e-13	1.63e-13	5.23e-11	6.71e-12

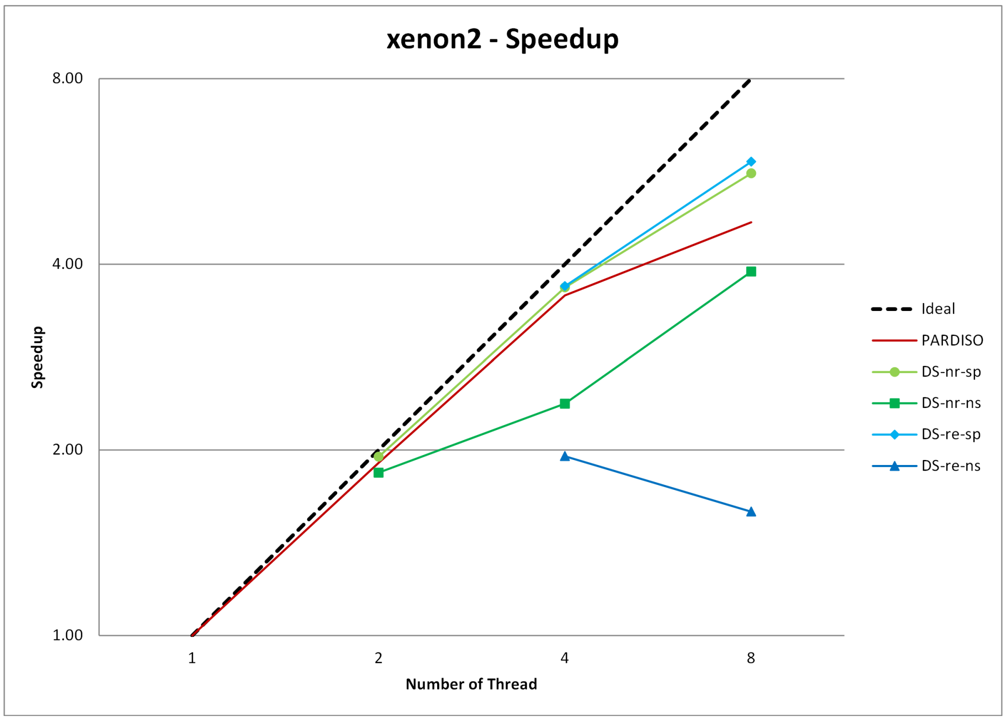

Figure 5.14: Matrix xenon2: Speedup versus Number of Threads

CHAPTER 6

CONCLUSION AND FUTURE WORK

For most of the real problems, solution of the linear systems of equations is a necessity. With the recent developments on the processor architectures it is also needed to distribute the workload to different threads or processes. However, most of the current LU factorization based parallel solvers have limits of parallelization since they need some sequential computations to be held.

Nonetheless, our DS factorization based parallel algorithm distribute the workload so that it does not need sequential computations or does not have data dependency for most of the steps as it is shown on Chapter 4. The algorithm proposed in this thesis, a new multi-threaded and recursive direct algorithm for parallel solution of sparse linear systems, is robust and works as fast as and for some cases faster than a well known parallel direct sparse linear solver, PARDISO on given matrices from different problem domains. Performance improvement is more pronounced for larger number of cores.

Using shared memory enables it to be used in personal computers and be enhanced with distributed memory or hybrid approach to get better results on high performance computers.

Recursion is used in the algorithm and our experiments show that it is more efficient for working on a diagonally dominant matrix. Another interesting output of our work is that we do not need to reorder the reduced system again before recursively applying our algorithm on it, since the reduced system has similar characteristics with the initial linear system.

Our experiments have also shown that using sparse right hand side feature on DS factorization results in a speed-boost without losing accuracy, although it means to skip the calculation of some non-zero values.

For the future, the following enhancements can be applied to achieve better results:

1) Recursion can also be used for the factorization phase of the algorithm if the diagonal blocks are diagonally dominant. Since the degree of diagonal dominance is determined by the minimum ratio obtained from all diagonal elements, a diagonal block of a matrix

could be diagonally dominant even if the whole matrix is not. Since our algorithm can also be implemented as a distributed memory algorithm, an option could be to solve the whole system of linear equations on different nodes of a high performance computer and execute the inner solutions with multi-threading.

2) Sparse right hand side feature of PARDISO has proven to be helpful because we expect that the different blocks of a sparse matrix would also be sparse. However, PARDISO uses dense structure as its input on right side of the linear system of equations. This results in inefficient usage of memory, i.e., allocation of dense arrays for sparse matrices. Another enhancement could be made by using or developing a sparse linear system solver that takes all inputs in sparse format. For this case, compressed sparse column (CSC) format could be useful.

3) Another change on factorization could be to make the reduced system even smaller. During the factorization phase, we calculate some values for the reduced system that are necessary for our computations on reduced system (light green cells on figure 4.9). However, these cells are affected by the degree of block diagonal dominance of the matrix. In other words, if the matrix is block diagonally dominant these values become practically zero. Another improvement can be to permute the matrix to increase its degree of block diagonally dominance and to skip the calculation of the values that are practically zero.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 8, 1965.

[2] C. T. Chen, *Linear system theory and design.* Oxford University Press, 1998.

[3] V. Kumar, A. Grama, A. Gupta, and Karypis, *Introduction to parallel computing.* Addison Wesley, second ed., 2003.

[4] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, Dec. 2011.

[5] E. S. Bölükbaşı and M. Manguoğlu, "A new multi-threaded and recursive direct algorithm for parallel solution of sparse linear systems." 5th International Conference of the ERCIM (European Research Consortium for Informatics and Mathematics) Working Group on Computing & Statistics, 2012.

[6] M. T. Heath, *Scientific Computing: An Introductory Survey.* New York: McGraw-Hill, 1997.

[7] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, "A supernodal approach to sparse partial pivoting," *SIAM J. Matrix Analysis and Applications*, vol. 20, no. 3, pp. 720–755, 1999.

[8] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," *ACM Trans. Mathematical Software*, vol. 31, pp. 302–325, September 2005.

[9] A. H. Sameh and D. J. Kuck, "On stable parallel linear system solvers," *Journal of the ACM (JACM)*, vol. 25, no. 1, pp. 81–91, 1978.

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

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

[12] M. Manguoğlu, "A domain-decomposing parallel sparse linear system solver," *Journal of computational and applied mathematics*, vol. 236, no. 3, pp. 319–325, 2011.

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

[14] B. F. Smith, "Domain decomposition methods for partial differential equations," in *Parallel Numerical Algorithms*, pp. 225–243, Springer Netherlands, 1997.

[15] P. B. Smith, Barry and W. Gropp, *Domain decomposition: parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, 2004.

[16] A. Toselli and O. B. Widlund, *Domain decomposition methods: algorithms and theory*, vol. 34. Sipringer Berlin, 2005.

[17] 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.

[18] M. Grote and H. Simon, "Parallel preconditioning and approximation inverses on the connection machine," in *Scalable High Performance Computing Conference, 1992. SHPCC-92, Proceedings.*, pp. 76–83, 1992.

[19] M. Benzi, "Preconditioning techniques for large linear systems: A survey," *Journal of Computational Physics*, vol. 182, no. 2, pp. 418–477, 2002.

[20] Y. Saad, *Iterative methods for sparse linear systems.* Siam, second ed., 2003.

[21] O. Østerby and Z. Zlatev, "Direct methods for sparse matrices," *DAIMI Report Series*, vol. 9, no. 123, 1980.

[22] T. A. Davis, *Direct methods for sparse linear systems.* Siam, 2006.

[23] P. Amestoy, I. Duff, J.-Y. L'Excellent, and J. Koster, "Mumps: A general purpose distributed memory sparse solver," in *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia* (T. Sørevik, F. Manne, A. Gebremedhin, and R. Moe, eds.), vol. 1947 of *Lecture Notes in Computer Science*, pp. 121–130, Springer Berlin Heidelberg, 2001.

[24] O. Schenk, M. Bollhöfer, and R. A. Römer, "On large-scale diagonalization techniques for the anderson model of localization," *SIAM review*, vol. 50, no. 1, pp. 91–112, 2008.

[25] O. Schenk, A. Wächter, and M. Hagemann, "Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization," *Computational Optimization and Applications*, vol. 36, no. 2-3, pp. 321–341, 2007.

[26] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with pardiso," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.

[27] O. Schenk and K. Gärtner, "On fast factorization pivoting methods for sparse symmetric indefinite systems," *Electronic Transactions on Numerical Analysis*, vol. 23, pp. 158–179, 2006.

[28] T. A. Davis, "A column pre-ordering strategy for the unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 165–195, 2004.

[29] T. A. Davis, "Algorithm 832: Umfpack v4. 3—an unsymmetric-pattern multifrontal method," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 2, pp. 196–199, 2004.

[30] T. A. Davis and I. S. Duff, "A combined unifrontal/multifrontal method for unsymmetric sparse matrices," *ACM Transactions on Mathematical Software (TOMS)*, vol. 25, no. 1, pp. 1–20, 1999.

[31] T. Davis and I. 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.

[32] W. Stallings, *Operating Systems: Internals and Design Principles, 6/E.* Prentice Hall, 2011.

[33] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts.* J. Wiley & Sons, 2009.

[34] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference.* Cambridge, MA, USA: MIT Press, 1995.

[35] Intel, "Intel cilk plus." `http://software.intel.com/en-us/intel-cilk-plus`, 2013.

[36] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*, vol. 10. The MIT Press, 2008.

[37] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, (Santa Barbara, California), pp. 207–216, July 1995.

[38] C. F. Joerg, *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, Jan. 1996. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-701.

[39] K. H. Randall, *Cilk: Efficient Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[40] P. Odifreddi, *Classical recursion theory: The theory of functions and sets of natural numbers.* Access Online via Elsevier, 1992.

[41] Intel, "Intel math kernel library 11.0." `http://software.intel.com/en-us/intel-mkl`, 2013.

[42] Intel, "Intel mkl 10.3 release notes." `http://software.intel.com/en-us/articles/intel-mkl-103-release-notes`, 2012.

[43] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide.* Philadelphia, PA: Society for Industrial and Applied Mathematics, third ed., 1999.

[44] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide.* Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[45] G. Karypis, “Metis - serial graph partitioning and fill-reducing matrix ordering.” `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`, 2013.

[46] HSL, “A collection of fortran codes for large scale scientific computation.” `http://www.hsl.rl.ac.uk`, 2011.

[47] Middle East Technical University Department of Computer Engineering, “High performance computing.” `http://www.ceng.metu.edu.tr/hpc/index`, 2011.

APPENDIX A

RESULTS OF ALL NUMERICAL EXPERIMENTS

The coding used for representing the results are as follows:

Reordering: The time consumed by METIS reordering in seconds.

Solution: The time consumed by solving the linear system of equation in seconds.

Exact Residual: The relative residual obtained from the exact solution.

Appx. Residual: The relative residual obtained from the approximate solution, i.e., dropping more elements using sparse right hand side feature of PARDISO.

DS-NR-SP: DS factorization without recursion and using the right hand side feature of PARDISO.

DS-NR-NS: DS factorization without recursion not using the right hand side feature of PARDISO.

DS-RE-SP: DS factorization with recursion and using the right hand side feature of PARDISO.

DS-RE-NS: DS factorization without recursion not using the right hand side feature of PARDISO.

A.1 ASIC_320k

Table A.1: PARDISO on ASIC_320k

Operation	**1 thread**	**2 threads**	**4 threads**	**8 threads**
Reordering	3.31	3.34	3.34	3.33
Solution	13.88	7.22	5.06	4.10
Exact Residual	1.29e-10	1.41e-10	9.03e-11	5.31e-11

Table A.2: DS-NR-SP on ASIC_320k

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	3.14	3.15	3.17
Solution	7.19	5.11	3.98
Exact Residual	1.12e-15	1.04e-15	9.98e-16
Appx. Residual	3.57e-09	9.43e-11	8.48e-09

Table A.3: DS-NR-NS on ASIC_320k

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	3.14	3.15	3.17
Solution	11.26	7.68	5.82
Exact Residual	5.63e-11	5.45e-11	1.16e-11

Table A.4: DS-RE-SP on ASIC_320k

Operation	**4 threads**	**8 threads**
Reordering	3.15	3.17
Solution	5.13	3.94
Exact Residual	1.47e-15	1.16e-15
Appx. Residual	1.01e-07	8.31e-08

Table A.5: DS-RE-NS on ASIC_320k

Operation	**4 threads**	**8 threads**
Reordering	3.15	3.17
Solution	7.67	4.14
Exact Residual	3.81e-13	3.84e-13

A.2 ASIC_620ks

Table A.6: PARDISO on ASIC_620ks

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	7.82	8.01	8.00	8.03
Solution	104.30	52.45	26.66	14.53
Exact Residual	9.45e-08	9.45e-08	9.45e-08	9.45e-08

Table A.7: DS-NR-SP on ASIC_620ks

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	7.44	7.47	7.47
Solution	52.41	26.27	13.92
Exact Residual	6.19e-10	8.41e-10	7.34e-10
Appx. Residual	1.17e-06	1.17e-06	4.20e-05

Table A.8: DS-NR-NS on ASIC_620ks

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	7.44	7.47	7.47
Solution	91.64	68.13	51.92
Exact Residual	8.26e-11	8.24e-11	8.26e-11

Table A.9: DS-RE-SP on ASIC_620ks

Operation	**4 threads**	**8 threads**
Reordering	7.47	7.47
Solution	27.19	14.81
Exact Residual	8.40e-10	8.13e-10
Appx. Residual	7.61e-05	2.03e-04

Table A.10: DS-RE-NS on ASIC_620ks

Operation	**4 threads**	**8 threads**
Reordering	7.47	7.47
Solution	69.26	52.17
Exact Residual	6.13e-11	6.18e-11

A.3 Crashbasis

Table A.11: PARDISO on crashbasis

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	2.32	2.42	2.41	2.68
Solution	3.74	1.98	1.12	1.04
Exact Residual	2.20e-15	2.20e-15	2.20e-15	2.19e-15

Table A.12: DS-NR-SP on crashbasis

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.67	1.32	2.71
Solution	1.91	1.17	0.87
Exact Residual	2.20e-15	2.15e-15	2.13e-15
Appx. Residual	8.64e-02	1.19e-01	1.75e-01

Table A.13: DS-NR-NS on crashbasis

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.67	1.32	2.71
Solution	2.03	1.34	1.02
Exact Residual	2.21e-15	2.14e-15	2.14e-15

Table A.14: DS-RE-SP on crashbasis

Operation	**4 threads**	**8 threads**
Reordering	1.32	2.71
Solution	0.96	0.51
Exact Residual	2.01e-14	2.44e-14
Appx. Residual	8.05e+00	1.16e+01

Table A.15: DS-RE-NS on crashbasis

Operation	**4 threads**	**8 threads**
Reordering	1.32	2.71
Solution	1.07	1.00
Exact Residual	2.13e-15	4.34e-16

A.4 Ecology2

Table A.16: PARDISO on ecology2

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	7.62	7.79	7.71	7.80
Solution	7.49	3.97	2.21	1.64
Exact Residual	1.32e-16	1.31e-16	1.31e-16	1.29e-16

Table A.17: DS-NR-SP on ecology2

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	2.19	4.39	8.78
Solution	3.91	2.03	1.28
Exact Residual	1.32e-16	1.31e-16	1.29e-16
Appx. Residual	7.28e-03	1.01e-02	1.52e-02

Table A.18: DS-NR-NS on ecology2

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	2.19	4.39	8.78
Solution	4.61	2.47	1.93
Exact Residual	1.37e-16	1.46e-16	1.45e-16

Table A.19: DS-RE-SP on ecology2

Operation	**4 threads**	**8 threads**
Reordering	4.39	8.78
Solution	1.96	0.99
Exact Residual	2.04e-16	1.71e-16
Appx. Residual	6.07e-01	7.14e-01

Table A.20: DS-RE-NS on ecology2

Operation	**4 threads**	**8 threads**
Reordering	4.39	8.78
Solution	2.29	1.74
Exact Residual	1.54e-16	1.54e-16

A.5 Freescale1

Table A.21: PARDISO on Freescale1

Operation	1 threads	2 threads	4 threads	8 threads
Reordering	25.42	25.61	25.48	25.73
Solution	7.64	4.15	2.52	2.27
Exact Residual	1.79e-10	1.79e-10	1.79e-10	1.85e-10

Table A.22: DS-NR-SP on Freescale1

Operation	2 threads	4 threads	8 threads
Reordering	24.11	25.71	26.04
Solution	3.99	2.15	1.56
Exact Residual	4.47e-15	4.29e-15	8.64e-16
Appx. Residual	1.26e-10	1.28e-10	1.35e-10

Table A.23: DS-NR-NS on Freescale1

Operation	2 threads	4 threads	8 threads
Reordering	24.11	25.71	26.04
Solution	8.64e-16	5.17	4.71
Exact Residual	2.11e-15	9.08e-16	8.61e-16

Table A.24: DS-RE-SP on Freescale1

Operation	4 threads	8 threads
Reordering	25.71	26.04
Solution	2.02	1.19
Exact Residual	7.11e-15	1.03e-15
Appx. Residual	1.06e-07	4.14e-06

Table A.25: DS-RE-NS on Freescale1

Operation	4 threads	8 threads
Reordering	25.71	26.04
Solution	5.64	4.91
Exact Residual	6.71e-16	1.94e-16

A.6 Hvdc2

Table A.26: PARDISO on hvdc2

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	0.99	1.02	1.00	0.95
Solution	0.45	0.35	0.29	0.23
Exact Residual	2.84e-09	2.80e-09	2.80e-09	2.85e-09

Table A.27: DS-NR-SP on hvdc2

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	1.09	1.09	1.15
Solution	-	0.03	0.01
Exact Residual	-	3.44e+01	1.73e+01
Appx. Residual	-	1.30e+03	1.89e+04

Table A.28: DS-NR-NS on hvdc2

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	1.09	1.09	1.15
Solution	-	0.07	0.04
Exact Residual	-	9.55e-11	1.40e-10

Table A.29: DS-RE-SP on hvdc2

Operation	**4 threads**	**8 threads**
Reordering	1.09	1.15
Solution	-	-
Exact Residual	-	-
Appx. Residual	-	-

Table A.30: DS-RE-NS on hvdc2

Operation	**4 threads**	**8 threads**
Reordering	1.09	1.15
Solution	0.08	0.04
Exact Residual	9.51e-11	1.24e-10

A.7 Kaufhold

Table A.31: PARDISO on Kaufhold

Operation	1 threads	2 threads	4 threads	8 threads
Reordering	0.06	0.05	0.06	0.07
Solution	0.04	0.03	0.02	0.02
Exact Residual	9.72e-14	9.72e-15	9.72e-15	9.72e-15

Table A.32: DS-NR-SP on Kaufhold

Operation	2 threads	4 threads	8 threads
Reordering	0.02	0.05	0.11
Solution	0.13	0.10	0.09
Exact Residual	1.26e-16	1.23e-16	1.20e-16
Appx. Residual	9.61e-10	1.27e-16	5.09e-03

Table A.33: DS-NR-NS on Kaufhold

Operation	2 threads	4 threads	8 threads
Reordering	0.02	0.05	0.11
Solution	0.13	0.10	0.10
Exact Residual	1.25e-16	1.23e-16	1.25e-16

Table A.34: DS-RE-SP on Kaufhold

Operation	4 threads	8 threads
Reordering	0.05	0.11
Solution	0.10	0.15
Exact Residual	3.14e-04	6.51e-16
Appx. Residual	2.61e-06	9.27e-05

Table A.35: DS-RE-NS on Kaufhold

Operation	4 threads	8 threads
Reordering	0.05	0.11
Solution	0.12	0.17
Exact Residual	1.93e-16	1.37e-16

A.8 Kuu

Table A.36: PARDISO on Kuu

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	0.07	0.06	0.06	0.08
Solution	0.06	0.04	0.03	0.03
Exact Residual	4.75e-16	4.59e-16	4.58e-16	4.51e-16

Table A.37: DS-NR-SP on Kuu

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.02	0.04	0.07
Solution	0.07	0.03	0.01
Exact Residual	7.41e-15	9.68e-15	1.81e-14
Appx. Residual	4.09e-10	9.74e-10	2.28e-09

Table A.38: DS-NR-NS on Kuu

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.02	0.04	0.07
Solution	0.20	0.11	0.08
Exact Residual	2.14e-16	2.13e-16	2.62e-16

Table A.39: DS-RE-SP on Kuu

Operation	**4 threads**	**8 threads**
Reordering	0.04	0.07
Solution	0.03	0.01
Exact Residual	3.85e-14	8.46e-14
Appx. Residual	1.17e-08	7.62e-07

Table A.40: DS-RE-NS on Kuu

Operation	**4 threads**	**8 threads**
Reordering	0.04	0.07
Solution	0.09	0.06
Exact Residual	2.56e-16	1.91e-16

A.9 Lin

Table A.41: PARDISO on Lin

Operation	1 threads	2 threads	4 threads	8 threads
Reordering	3.1	3.1	2.7	3.2
Solution	81.5	42.8	24.2	24.3
Exact Residual	9.95e-10	8.81e-16	8.07e-16	7.16e-16

Table A.42: DS-NR-SP on Lin

Operation	2 threads	4 threads	8 threads
Reordering	0.8	1.3	2.9
Solution	87.8	30.3	17.7
Exact Residual	5.63e-16	8.72e-16	1.36e-15
Appx. Residual	8.48e-11	1.28e-10	1.68e-10

Table A.43: DS-NR-NS on Lin

Operation	2 threads	4 threads	8 threads
Reordering	0.8	1.3	2.9
Solution	293.6	146.5	112.7
Exact Residual	1.88e-16	6.99e-16	2.14e-16

Table A.44: DS-RE-SP on Lin

Operation	4 threads	8 threads
Reordering	1.3	2.9
Solution	31.5	13.2
Exact Residual	3.70e-13	6.13e-13
Appx. Residual	8.63e-09	1.13e-08

Table A.45: DS-RE-NS on Lin

Operation	4 threads	8 threads
Reordering	1.3	2.9
Solution	117.4	81.5
Exact Residual	1.18e-12	8.65e-14

A.10 Majorbasis

Table A.46: PARDISO on majorbasis

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	2.31	2.31	2.30	2.30
Solution	14.50	7.64	4.24	3.29
Exact Residual	1.72e-15	1.71e-15	1.71e-15	1.70e-15

Table A.47: DS-NR-SP on majorbasis

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	1.26	2.03	2.71
Solution	7.29	4.01	2.84
Exact Residual	1.09e-16	1.09e-16	1.04e-16
Appx. Residual	1.17e-4	5.19e-13	8.23e-15

Table A.48: DS-NR-NS on majorbasis

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	1.26	2.03	2.71
Solution	11.72	8.08	5.91
Exact Residual	1.09e-16	1.09e-16	2.76e-16

Table A.49: DS-RE-SP on majorbasis

Operation	**4 threads**	**8 threads**
Reordering	2.03	2.71
Solution	4.02	2.75
Exact Residual	9.61e-05	2.91e-11
Appx. Residual	5.66e-03	4.11e-02

Table A.50: DS-RE-NS on majorbasis

Operation	**4 threads**	**8 threads**
Reordering	2.03	2.71
Solution	10.04	14.26
Exact Residual	2.78e-16	1.65e-16

A.11 Pd

Table A.51: PARDISO on Pd

Operation	1 threads	2 threads	4 threads	8 threads
Reordering	0.05	0.05	0.04	0.05
Solution	0.02	0.02	0.02	0.02
Exact Residual	2.38e-14	2.38e-14	2.38e-14	2.38e-14

Table A.52: DS-NR-SP on Pd

Operation	2 threads	4 threads	8 threads
Reordering	0.01	0.13	0.13
Solution	-	0.06	0.05
Exact Residual	-	2.38e-14	2.38e-14
Appx. Residual	-	7.14e+01	4.91e+03

Table A.53: DS-NR-NS on Pd

Operation	2 threads	4 threads	8 threads
Reordering	0.01	0.13	0.13
Solution	-	0.12	0.08
Exact Residual	-	1.64e-14	1.31e-14

Table A.54: DS-RE-SP on Pd

Operation	4 threads	8 threads
Reordering	0.13	0.13
Solution	-	-
Exact Residual	-	-
Appx. Residual	-	-

Table A.55: DS-RE-NS on Pd

Operation	4 threads	8 threads
Reordering	0.13	0.13
Solution	-	-
Exact Residual	-	-

A.12 Powersim

Table A.56: PARDISO on powersim

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	0.08	0.09	0.09	0.09
Solution	0.04	0.02	0.02	0.02
Exact Residual	9.42e-13	8.62e-13	9.55e-13	1.01e-12

Table A.57: DS-NR-SP on powersim

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	1.11	1.12	1.14
Solution	-	0.01	0.00
Exact Residual	-	2.26e-05	1.01e-06
Appx. Residual	-	1.17e-03	2.71e-02

Table A.58: DS-NR-NS on powersim

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	1.11	1.12	1.14
Solution	-	0.01	0.01
Exact Residual	-	6.02e-15	2.81e-15

Table A.59: DS-RE-SP on powersim

Operation	**4 threads**	**8 threads**
Reordering	1.12	1.14
Solution	-	-
Exact Residual	-	-
Appx. Residual	-	-

Table A.60: DS-RE-NS on powersim

Operation	**4 threads**	**8 threads**
Reordering	1.12	1.14
Solution	0.01	0.00
Exact Residual	6.11e-15	2.73e-15

A.13 Raj1

Table A.61: PARDISO on Raj1

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	1.90	1.84	1.90	1.83
Solution	0.84	0.46	0.30	0.19
Exact Residual	5.48e-09	6.16e-10	4.50e-10	6.47e-10

Table A.62: DS-NR-SP on Raj1

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.81	1.74	3.75
Solution	0.51	0.34	0.21
Exact Residual	4.67e-07	1.21e-08	1.04e-08
Appx. Residual	6.21e+05	7.40e+04	2.09e+04

Table A.63: DS-NR-NS on Raj1

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.81	1.74	3.75
Solution	1.12	0.84	0.68
Exact Residual	8.19e-08	1.00e-08	6.23e-09

Table A.64: DS-RE-SP on Raj1

Operation	**4 threads**	**8 threads**
Reordering	1.74	3.75
Solution	0.37	0.23
Exact Residual	3.04e+00	4.31e+00
Appx. Residual	4.28e+07	2.61e+08

Table A.65: DS-RE-NS on Raj1

Operation	**4 threads**	**8 threads**
Reordering	1.74	3.75
Solution	1.28	1.49
Exact Residual	1.13e-08	7.07e-09

A.14 Rajat21

Table A.66: PARDISO on rajat21

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	3.12	3.15	3.17	3.16
Solution	1.05	0.56	0.35	0.63
Exact Residual	2.54e-05	7.97e-06	8.38e-06	1.16e-05

Table A.67: DS-NR-SP on rajat21

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	3.04	3.06	3.11
Solution	0.59	0.37	0.24
Exact Residual	4.93e-07	2.65e-07	8.04e-08
Appx. Residual	2.19e-01	1.05e-01	7.28e-02

Table A.68: DS-NR-NS on rajat21

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	3.04	3.06	3.11
Solution	0.91	0.76	0.85
Exact Residual	2.51e-07	1.11e-07	5.68e-08

Table A.69: DS-RE-SP on rajat21

Operation	**4 threads**	**8 threads**
Reordering	3.06	3.11
Solution	0.39	0.28
Exact Residual	3.18e-04	9.96e-05
Appx. Residual	7.01e+03	1.94e+03

Table A.70: DS-RE-NS on rajat21

Operation	**4 threads**	**8 threads**
Reordering	3.06	3.11
Solution	0.89	1.61
Exact Residual	1.18e-07	5.81e-08

A.15 Scircuit

Table A.71: PARDISO on scircuit

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	0.82	0.84	0.83	0.87
Solution	1.02	0.96	0.67	0.80
Exact Residual	1.93e-09	2.03e-09	1.60e-09	1.59e-09

Table A.72: DS-NR-SP on scircuit

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.78	0.78	0.78
Solution	1.51	1.03	1.01
Exact Residual	4.63e-15	9.94e-15	8.92e-15
Appx. Residual	3.20e-09	8.70e-09	1.33e-08

Table A.73: DS-NR-NS on scircuit

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.78	0.78	0.78
Solution	1.83	1.70	1.64
Exact Residual	9.94e-15	9.97e-15	8.90e-15

Table A.74: DS-RE-SP on scircuit

Operation	**4 threads**	**8 threads**
Reordering	0.78	0.78
Solution	1.11	1.05
Exact Residual	1.73e-15	1.44e-15
Appx. Residual	8.70e-09	1.33e-08

Table A.75: DS-RE-NS on scircuit

Operation	**4 threads**	**8 threads**
Reordering	0.78	0.78
Solution	1.73	2.05
Exact Residual	1.00e-14	9.95e-15

A.16 Stomach

Table A.76: PARDISO on stomach

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	3.35	3.35	3.34	3.21
Solution	10.73	5.51	3.02	2.11
Exact Residual	7.74e-16	7.71e-16	7.64e-16	7.52e-16

Table A.77: DS-NR-SP on stomach

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.61	1.22	2.56
Solution	5.48	2.86	1.87
Exact Residual	7.65e-16	8.17e-16	9.02e-16
Appx. Residual	1.10e-03	2.01e-03	2.19e-03

Table A.78: DS-NR-NS on stomach

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.61	1.22	2.56
Solution	5.47	3.54	2.16
Exact Residual	4.21e-16	4.13e-16	4.38e-16

Table A.79: DS-RE-SP on stomach

Operation	**4 threads**	**8 threads**
Reordering	1.22	2.56
Solution	2.84	1.85
Exact Residual	3.20e-15	3.27e-15
Appx. Residual	5.18e-01	1.14e+00

Table A.80: DS-RE-NS on stomach

Operation	**4 threads**	**8 threads**
Reordering	1.22	2.56
Solution	3.47	2.21
Exact Residual	3.96e-16	4.08e-16

A.17 Tomography

Table A.81: PARDISO on tomography

Operation	1 threads	2 threads	4 threads	8 threads
Reordering	0.01	0.02	0.02	0.03
Solution	0.02	0.01	0.02	0.01
Exact Residual	2.90e-13	3.72e-13	4.18e-13	3.67e-13

Table A.82: DS-NR-SP on tomography

Operation	2 threads	4 threads	8 threads
Reordering	0.09	0.09	4.36
Solution	0.02	0.01	0.01
Exact Residual	9.49e-13	8.82e-13	5.65e-13
Appx. Residual	1.41e+01	1.45e+01	9.71e+00

Table A.83: DS-NR-NS on tomography

Operation	2 threads	4 threads	8 threads
Reordering	0.09	0.09	4.36
Solution	0.03	0.02	0.02
Exact Residual	7.12e-13	5.37e-13	3.90e-13

Table A.84: DS-RE-SP on tomography

Operation	4 threads	8 threads
Reordering	0.09	4.36
Solution	0.04	0.05
Exact Residual	4.18e+00	1.09e+01
Appx. Residual	7.53e+05	5.16e+05

Table A.85: DS-RE-NS on tomography

Operation	4 threads	8 threads
Reordering	0.09	4.36
Solution	0.04	0.07
Exact Residual	2.05e-12	1.71e+01

A.18 Torso3

Table A.86: PARDISO on torso3

Operation	1 threads	2 threads	4 threads	8 threads
Reordering	4.99	4.65	4.66	4.62
Solution	59.32	31.34	16.78	18.79
Exact Residual	1.03e-15	1.02e-15	1.01e-15	9.93e-16

Table A.87: DS-NR-SP on torso3

Operation	2 threads	4 threads	8 threads
Reordering	1.19	1.86	4.13
Solution	30.28	15.91	11.37
Exact Residual	4.80e-15	1.74e-15	6.82e-16
Appx. Residual	9.61e-02	2.18e-01	3.65e-01

Table A.88: DS-NR-NS on torso3

Operation	2 threads	4 threads	8 threads
Reordering	1.19	1.86	4.13
Solution	42.08	22.74	17.93
Exact Residual	4.24e-15	1.69e-15	7.83e-16

Table A.89: DS-RE-SP on torso3

Operation	4 threads	8 threads
Reordering	1.86	4.13
Solution	16.21	19.06
Exact Residual	2.11e-15	8.59e-16
Appx. Residual	1.23e+00	7.01e+01

Table A.90: DS-RE-NS on torso3

Operation	4 threads	8 threads
Reordering	1.86	4.13
Solution	28.61	34.77
Exact Residual	1.42e-15	4.15e-16

A.19 Transient

Table A.91: PARDISO on transient

Operation	1 threads	2 threads	4 threads	8 threads
Reordering	1.18	1.18	1.21	1.23
Solution	0.43	0.24	0.16	0.11
Exact Residual	8.88e-07	1.42e-06	1.56e-06	1.68e-06

Table A.92: DS-NR-SP on transient

Operation	2 threads	4 threads	8 threads
Reordering	1.19	1.19	1.19
Solution	0.29	0.22	0.18
Exact Residual	1.96e-10	1.80e-10	1.47e-10
Appx. Residual	3.51e-06	7.24e-06	1.82e-05

Table A.93: DS-NR-NS on transient

Operation	2 threads	4 threads	8 threads
Reordering	1.19	1.19	1.19
Solution	0.49	0.50	0.49
Exact Residual	4.17e-10	4.07e-10	3.82e-10

Table A.94: DS-RE-SP on transient

Operation	4 threads	8 threads
Reordering	1.19	1.19
Solution	0.16	0.12
Exact Residual	2.41e-03	2.41e-03
Appx. Residual	1.7e+01	8.38e+02

Table A.95: DS-RE-NS on transient

Operation	4 threads	8 threads
Reordering	1.19	1.19
Solution	0.67	0.71
Exact Residual	7.07e-11	5.94e-11

A.20 Xenon2

Table A.96: PARDISO on xenon2

Operation	**1 threads**	**2 threads**	**4 threads**	**8 threads**
Reordering	1.81	1.83	1.75	1.74
Solution	15.95	8.37	4.48	3.41
Exact Residual	4.38e-12	4.39e-12	4.37e-12	4.35e-12

Table A.97: DS-NR-SP on xenon2

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.74	1.00	2.23
Solution	8.18	4.35	2.84
Exact Residual	9.41e-12	4.33e-12	1.74e-13
Appx. Residual	9.16e-08	4.97e-08	1.01e-09

Table A.98: DS-NR-NS on xenon2

Operation	**2 threads**	**4 threads**	**8 threads**
Reordering	0.74	1.00	2.23
Solution	8.68	6.71	4.10
Exact Residual	1.12e-13	1.12e-13	1.63e-13

Table A.99: DS-RE-SP on xenon2

Operation	**4 threads**	**8 threads**
Reordering	1.00	2.23
Solution	4.33	2.72
Exact Residual	2.21e-11	5.23e-11
Appx. Residual	3.29e-06	7.08e-06

Table A.100: DS-RE-NS on xenon2

Operation	**4 threads**	**8 threads**
Reordering	1.00	2.23
Solution	8.16	10.04
Exact Residual	6.71e-12	6.71e-12