PARALLEL PRECONDITIONING TECHNIQUES FOR NUMERICAL SOLUTION OF THREE DIMENSIONAL PARTIAL DIFFERENTIAL EQUATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ABDULLAH ALİ SİVAS

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
SCIENTIFIC COMPUTING

JULY 11TH, 2016

Approval of the thesis:

PARALLEL PRECONDITIONING TECHNIQUES FOR NUMERICAL SOLUTION OF THREE DIMENSIONAL PARTIAL DIFFERENTIAL EQUATIONS

submitted by **ABDULLAH ALİ SİVAS** in partial fulfillment of the requirements for the degree of **Master of Science in Department of Scientific Computing, Middle East Technical University** by,

Prof. Dr. Bülent Karasözen
Director, Graduate School of **Applied Mathematics** _______________

Assoc. Prof. Dr. Ömür Uğur
Head of Department, **Scientific Computing** _______________

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

Examining Committee Members:

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

Prof. Dr. Hasan U. Akay
Department of Mechanical Engineering, Atilim University _______________

Assoc. Prof. Dr. Ömür Uğur
Scientific Computing, METU _______________

Date: _______________

ABSTRACT

PARALLEL PRECONDITIONING TECHNIQUES FOR NUMERICAL SOLUTION OF THREE DIMENSIONAL PARTIAL DIFFERENTIAL EQUATIONS

Sivas, Abdullah Ali
M.S., Department of Scientific Computing
Supervisor : Assoc. Prof. Dr. Murat Manguoğlu

July 11th, 2016, 42 pages

Partial differential equations are commonly used in industry and science to model observed phenomena and gain insight regarding phenomena or solve related problems. Recently three dimensional partial differential equations started to become more and more essential and popular. Numerical solution of these problems generally is composed of two steps; discretization and solving resulting sparse linear systems which are large and usually ill-conditioned. For large three dimensional problems, it is imperative to use iterative solvers rather than direct solvers due to small memory requirement and short solution times, but iterative solvers mostly fail for ill-conditioned coefficient matrices hence they are not as robust as direct solvers without an effective preconditioner. Preconditioning is a remedy for this problem. Solution of large sparse linear systems require large amounts of time and usually solution of multiple linear systems is necessary. Parallel computing techniques are used to overcome this problem. Various preconditioning techniques with iterative techniques and their scalability on three different parallel computing platforms are investigated for the solution of two three dimensional partial differential equations arising in large scale problems which are important for industrial applications and scientific modelling. Results of this investigation are also compared against the state-of-the-art direct solvers.

Keywords : numerical linear algebra, preconditioner, sparse approximate inverse, high performance computing, parallel computing

ÖZ

ÜÇ BOYUTLU KISMİ DİFERANSİYEL DENKLEMLERİN NÜMERİK
ÇÖZÜMÜ İÇİN PARALEL ÖNKOŞULLANDIRMA TEKNİKLERİ

Sivas, Abdullah Ali
Yüksek Lisans, Bilimsel Hesaplama Bölümü
Tez Yöneticisi : Doç. Dr. Murat Manguoğlu

11 Temmuz 2016, 42 sayfa

Kısmi diferansiyel denklemler genellikle gözlenen olguları modelleme ve bu olgularla ilgili içgörü kazanmak veya ilgili sorunları çözmek için sanayi uygulamaları ve bilimde kullanılmaktadır. Son zamanlarda üç boyutlu kısmi diferansiyel denklemler daha önemli ve popüler olmaya başlamıştır. Bu sorunların sayısal çözümü genellikle problemin seyrek doğrusal sistem çözümü problemine dönüştürülmesi ve çoğunlukla büyük ve kötü koşullandırılmış bu doğrusal sistemlerin çözülmesinden oluşur. Büyük boyu nedeniyle doğrudan çözücüler yerine küçük bellek gereksinimi ve kısa çözüm süreleriyle iteratif çözücüler çekici gelse de, yinelemeli çözücüler çoğunlukla kötü koşullandırılmış doğrusal sistemleri çözememektedirler dolayisiyla doğrudan çözücüler kadar gürbüz değillerdir. Önkoşullandırma bu sorun için bir çaredir. Büyük seyrek doğrusal sistemlerinin çözümü büyük miktarda zaman gerektirmektedir ve genellikle problemin çözümü için pek çok dogrusal sistemin çözümü gereklidir. Paralel hesaplama teknikleri bu sorunun üstesinden gelmek için kullanılır. Çeşitli önkoşullandırma teknikleri iteratif çözüm yöntemleriyle sanayi uygulamaları ve bilimsel modelleme açısından önemli olan iki kısmi diferansiyel denklemin çözümü için incelenmiştir. Bu araştırmanın sonuçları, en modern doğrudan çözücüler ile karşılaştırılmıştır.

Anahtar Kelimeler : nümerik lineer cebir, önkoşullandıcılar, seyrek yaklaşık ters, yüksek başarımlı hesaplama, paralel hesaplama

To everyone who supported me

ACKNOWLEDGMENTS

I would like to express my great appreciation of my thesis advisor Assoc. Prof. Dr. Murat Manguoğlu. His guidance and support strongly affected my class and thesis performance in the best way. During my studies with him, I have learned many interesting things which I was interested in but not accustommed to. I also want to thank Prof. Dr. Bülent Karasözen for suggesting me Scientific Computing program and introducing me to my thesis advisor. I also want to acknowledge his patient lecturing during implementation of three dimensional extension of finite volume schemes. My thesis studies started with a problem by Dr.ir. J.H.M. ten Thije Boonkkamp and Dr.ir. Martijn Anthonissen. They have patiently answered all my questions about the problems and this study resulted with a presentation in ENUMATH 2015. Also during my studies, I got to know Prof. Dr. Hasan Akay and Dr. Erdal Oktay. Both of them taught me a lot, gave me important perspectives and I enjoyed working with them. They both have great knowledge and great ideas to improve themselves, their students and, in general, industry and science. Last but not least, I want to thank everyone in the Institute of Applied Mathematics, for nice and friendly environment they provided.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Linear systems of equations are frequently encountered in linear programming, optimization, computational fluid dynamics, etc. Solution of linear system of equations is an essential part in solution of these problems. Main challenges are large number unknowns and ill-conditioning of the coefficient matrix. Large number of unknowns implies the need for large amounts of memory. If the coefficient matrix is sparse, necessary amount is less (since number of nonzeros is small with respect to number of zeroes and zeros are not stored explicitly) as well as amount of operations done, as we do not perform arithmetic operation with zeros.

Numerical solution of partial differential equations (PDEs) usually requires the solution of a large sparse linear system of equations. Direct methods and iterative methods are two commonly used methods. Direct methods, even though robust, are slow and memory consuming if fill-in occurs. Iterative methods are not as robust as direct methods but are faster and require much less memory. With increased availability of multi-core processors and clusters it is inevitable to come up with scalable techniques.

Parallel scalability is measured in terms of the speedup as the number of processing elements are increased. Ideally, for example, when an eight-core computer used, a method should complete same work at 1/8 of the time it requires when a single-core computer is used. Some large scale applications take quite long times which can be reduced to a fraction using scalable algorithms and parallel computing platforms. Therefore, scalability of a method is important for time sensitive, industrial applications and scientific modelling.

In this thesis, we have investigated iterative methods and preconditioning technigues in the context of two important PDE-related problems. Direct methods are used as a robust baseline for comparison. Different iterative methods are used to suit requirements of PDEs. Various preconditioners are applied and their scalability and effect on performance are examined.

Two application problems that we are focused on are topology optimization and three dimensional advection-diffusion-reaction (ADR) equation. Both problems are important and has many applications. For example topology optimization is used for designing structures that are optimized for performance and at time minimized the material being used. This not only saves the production costs but also minimizes the negative effects on the enviroment. ADR equations arise in plasma physics, fluid dynamics, combustion theory and in many more areas. Fast and high precision solution of these equations are important for simulations as actual experiments which may be expensive, time consuming and sometimes even dangerous. These simulations give crucial insights for modelled phenomena.

Results obtained in this thesis are (accepted) to be published in [1, 24].

The content of this thesis is summarized as; in Chapter 2 solution methods and preconditioners for sparse linear system of equations are explained and parallelization of the methods are given, in Chapter 3 application problems are elaborated and challenges in solving them are mentioned. Later in Chapter 4 results of numerical experiments are presented and discussed and in Chapter 5 conclusions are given with possible future work.

CHAPTER 2

SOLUTION METHODS FOR LARGE SPARSE LINEAR SYSTEMS

In this chapter, we explain some methods for solving a specific class of linear systems. We can, among other possible ways, classify linear systems of equations in two groups; dense and sparse. In the following, we will consider solution methods for the sparse linear systems.

Solving large sparse linear system of equations is a challenging problem. Historically, the main difficulty is finding a balance between robustness, time and space complexities. With the introduction of multi/many-core processors, parallel scalability of the algorithm also plays an important role.

A sparse linear system contains substantially less nonzeros elements compared to zeros. We can take advantage of this by avoiding multiplication by zeros and summation by zeros. Therefore, solving large sparse linear systems greatly differs from solving dense or full linear systems in terms of performance and time required. Even more performance gain can be achieved by exploiting further properties of the linear system, such as symmetry, bandedness and positive definiteness.

There are two main classes of solvers, namely direct and iterative solvers for sparse linear systems. It is well known that direct solvers are robust but have higher time and space complexity and their parallel scalability is often limited especially for systems that arise from three dimensional problems. This is mainly because of the fill-in that is generated during the factorization. Iterative solvers are more prone to error than direct solvers but they require less memory and short solution times provided an effective preconditioner is used. Their parallel scalability is also better than direct solvers, again if used together with a scalable preconditioner.

Main difference between direct and iterative solution methods is robustness. Except error from finite precision arithmetic, direct solution methods are guarenteed to find the solution if one exists. Iterative solution methods are in some cases may not find any solution (divergence) or find a solution which is wrong. This is due to conditioning of the linear system. In cases which system is ill-conditioned, to solve linear system with an iterative method, it is possible to precondition the system. It introduces another cost (computation and application of preconditioner), but in many cases it is still preferable over direct solution methods.

Stationary and non-stationary iterative methods are two main classes of iterative solvers. They are further divided into many sub-classes. In practice, Krylov subspace solvers, which are non-stationary methods, are often used. Some examples of Krylov subspace methods include, Conjugate Gradient (CG), Generalized Minimum Residual (GMRES), Bi-Conjugate Gradient Stabilized (BiCGStab), Quasi-Minimum Residual (QMR) and others. The list of possible Krylov subspace algorithms that can be used (see for example [6]) can narrowed down depending on the properties of the coefficient matrix. Even then, there are still some decisions that needs to be made and it might get complicated if one takes parallel scalability into account as well.

In addition to that, using an iterative scheme without a preconditioner would not be practical as it will require many iterations and hence an effective preconditioner is needed. The preconditioner is chosen so that the preconditioned system has a more favorable eigenvalue distribution and hence the number of iterations to convergence is reduced. The performance of three parallel preconditioners, that are based on the sparse approximate inverse (SAI) and incomplete LU (ILU) factorization and also Block-Jacobi preconditioner, are compared and studied.

2.1 Sparse Linear Systems

Sparse linear systems are vaguely defined as linear systems which have substantially less nonzeros compared to zeros in coefficient matrix. Usually this ratio is quite small in systems resulting from solution of PDEs, so they are sparse. If a matrix is sparse, we can take advantage of large number of zeros, for example by avoiding operations like multiplication and summation by zeros. Another advantage is that, we do not have to store zeros explicitly. Storing nonzeros and their locations is enough as any value that is not stored is obviously zero. Therefore sparse matrices hold much less memory space. Compatibility is a challenge when designing storage methods for sparse matrices as some designs may prevent efficient implementation of commonly used operations, like solution of linear systems. We now outline two of common storage schemes for sparse matrices.

2.1.1 Coordinate Format(COO)

This is the simplest storage format for sparse matrices. Basically matrix is stored as three arrays; first array stores the row numbers of nonzeros, second array stores the column numbers of nonzeros and third array stores the values of nonzeros. Lengths of all arrays are equal to the number of nonzeros. For example, consider;

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 3 & 0 & 4 & 0 \\ 9 & 0 & 5 & 0 & 0 \\ 0 & 8 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 7 \end{pmatrix}. \tag{2.1}$$

COO representation of this matrix is;

```
I = [1  1  2  2  3  3  4  4  5]
J = [1  3  2  4  1  3  2  4  5]
V = [1  2  3  4  9  5  8  6  7]
```

where `I` holds the row numbers, `J` holds the column numbers and `V` holds the values. Note that all arrays are of length 9, which is the number of nonzeros and matrix $\boldsymbol{A}$ is of size 5x5.

2.1.2 Compressed Sparse Row Format(CSR)

Compressed sparse row format, sometimes also called as Yale format, employs a simple idea. If matrix values are listed in row order, storage for array of the row numbers can be compressed. If number of nonzeros is large this may save a lot of space. As we know that the values are ordered by their row numbers, i.e. nonzeros on first row are stored in first part of array `V`, rather than holding row numbers we can somehow hold how many nonzeros are there on that row.

A sparse matrix is stored using three arrays in CSR format; first array contains pointers to where a row begins in other two arrays, second array contains column numbers and third array contains values. Again considering the matrix $\boldsymbol{A}$ (2.1), its CSR representation is;

```
IA = [1  3  5  7  8  9]
JA = [1  3  2  4  1  3  2  4  5]
VA = [1  2  3  4  9  5  8  6  7]
```

where `IA` holds the pointers, `JA` holds the column numbers and `VA` holds the values. Length of `IA` is 6 while rest is of length 9. Last 9 in array `IA` is to point to the end of the matrix and saving number of nonzeros at last row which is `IA(6)-IA(5)= 9-8= 1`.

2.2 Direct Solution of Sparse Linear Systems

Direct solution methods for sparse linear systems mostly make use of LU factorization. These methods try to minimize fill-in to save memory as much as possible. Fill-ins are the nonzero elements introduced during factorization process which were zeros to begin with. As zeros are not explicitly stored, introducing new nonzeros may be a complicated task depending on storage scheme. For example, if the coefficient matrix is stored in COO format, introducing a new nonzero is just adding a new element to each of the arrays and sorting them. Even though it seems very simple, dynamically reallocating the memory is costly. Guessing the amount of fill-ins and allocating memory on that guess is the usual way to avoid dynamic reallocation at any point. Introducing fill-in in CSR format is more complicated, as arithmetic operations must be done on row array and sorting of column and values arrays.

Usually direct solution of sparse linear systems consists of four parts. These parts are mentioned but not elaborately explained as they are out of scope of this thesis. First of all system is reordered to minimize fill-in and to enhance parallelism. Later, LU-factorization (symbolic factorization followed by the numerical factorization) is done. At this part, pivoting may be necessary to prevent zeros on the diagonals of L and U. Third part is the where linear system solved by using forward and backward triangular sweeps. Last part is optional, if $||\boldsymbol{b} - \boldsymbol{Ax}||$ is not small enough, iterative refinement can be applied to increase precision of solution.

2.3 Iterative Solution of Sparse Linear Systems

There are many variations of iterative solvers. Historically, they were based on relaxation of coordinates. Main idea is to start with an initial guess and update it to eliminate one or few nonzeros in residual vector $\boldsymbol{r} = ||\boldsymbol{b} - \boldsymbol{Ax}||$. Examples are Jacobi, Gauss-Seidel and SOR. These are historically important but rarely preferred in practice. Commonly used methods uses some kind of projection process. These methods aims to find an approximation to solution from a subspace. If this subspace is chosen to be a Krylov subspace, these methods are called as Krylov subspace methods. Algorithms given in this chapter are from [22].

Most of the linear systems resulting from solution of PDEs have large condition numbers which may result in divergence, false convergence or failure of iterative methods. Even though none of these happen, convergence may be slow. Preconditioning is essential to enhance the parallel performance and the robustness of iterative methods.

2.3.1 Projection Methods

Consider the linear system;

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{2.2}$$

where $\boldsymbol{A}$ is an $n \times n$ matrix. Projection methods try to find an approximate solution to the linear system from a subspace of $\mathbb{R}^n$. Let $\mathcal{K}$ be the search subspace and m be its dimension. Some constraints must be imposed to extract approximation, these constraints are usually described with m independent orthogonal vectors. These vectors define another subspace $\mathcal{L}$ which is called subspace of constraints. Constraints are imposed with the condition $\boldsymbol{b} - \boldsymbol{Ax} \perp \mathcal{L}$. This idea is widely known as Petrov-Galerkin conditions and commonly used in various mathematical methods.

There are two choices for $\mathcal{K}$ and $\mathcal{L}$ which results in different classes of projection methods. If $\mathcal{K} = \mathcal{L}$ then methods is called orthogonal projection method, otherwise it is called oblique projection method.

Now, again consider the linear system above and let $\mathcal{K}$ and $\mathcal{L}$ be subspaces of $\mathbb{R}^n$. Projection methods try to solve below problem;

$$\text{Find } \widetilde{x} \in \mathcal{K}, \text{ such that } \boldsymbol{b} - \boldsymbol{A}\widetilde{\boldsymbol{x}} \perp \mathcal{L}. \tag{2.3}$$

Provided initial guess x_0 is given, then instead of $\mathcal{K}$, $x_0 + \mathcal{K}$ should be used. Then problem turns into;

$$\text{Find } \widetilde{x} \in x_0 + \mathcal{K}, \text{ such that } \boldsymbol{b} - \boldsymbol{A}\widetilde{\boldsymbol{x}} \perp \mathcal{L}. \tag{2.4}$$

Approximate solution can be written as;

$$\widetilde{x} = x_0 + \delta, \quad \delta \in \mathcal{K}, \tag{2.5}$$

$$(\boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x_0} + \boldsymbol{\delta}), \boldsymbol{\omega}) = 0, \quad \forall \omega \in \mathcal{L}. \tag{2.6}$$

Rather than using subspaces themselves we can use matrices whose column vectors form bases for these subspaces. This greatly helps to derive algorithms. Let $\boldsymbol{V}$ be matrix representation of $\mathcal{K}$ and $\boldsymbol{W}$ be matrix representation of $\mathcal{L}$. Then approximate solution is;

$$\widetilde{\boldsymbol{x}} = \boldsymbol{x_0} + \boldsymbol{V}\boldsymbol{y}. \tag{2.7}$$

Imposing orthogonality conditions, $\omega \in \boldsymbol{W}$,

$$0 = (\boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x_0} + \boldsymbol{\delta}), \boldsymbol{\omega}) \tag{2.8}$$

$$= \boldsymbol{W}^T\boldsymbol{b} - \boldsymbol{W}^T\boldsymbol{A}(\boldsymbol{x_0} + \boldsymbol{V}\boldsymbol{y}), \tag{2.9}$$

we get following linear system equations for $\boldsymbol{y}$;

$$\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{V}\boldsymbol{y} = \boldsymbol{W}^T\boldsymbol{b} - \boldsymbol{W}^T\boldsymbol{A}\boldsymbol{x_0}. \tag{2.10}$$

Assuming $\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{V}$ is non-singular, it can be shown that;

$$\widetilde{\boldsymbol{x}} = \boldsymbol{x_0} + \boldsymbol{V}(\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{V})^{-1}(\boldsymbol{W}^T\boldsymbol{b} - \boldsymbol{W}^T\boldsymbol{A}\boldsymbol{x_0}). \tag{2.11}$$

Now, a prototype algorithm can be given as;

```
while not converged do
    Choose subspaces $\mathcal{K}$ and $\mathcal{L}$
    Determine bases $\boldsymbol{V}$ and $\boldsymbol{W}$ for $\mathcal{K}$ and $\mathcal{L}$
    $\boldsymbol{r} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}$
    $\boldsymbol{y} := (\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{V})^{-1}\boldsymbol{W}^T\boldsymbol{r}$
    $\boldsymbol{x} := \boldsymbol{x} + \boldsymbol{V}\boldsymbol{y}$
end
```

Algorithm 1: Prototype Projection Method

Notice that algorithm above only works if $\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{V}$ is non-singular. This may not be guaranteed even if $\boldsymbol{A}$ is non-singular. But it can be easily seen that if no vectors in $\boldsymbol{A}\mathcal{K}$ is orthogonal to $\mathcal{L}$, $\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{V}$ is non-singular.

There are two important cases where $\boldsymbol{W}^T\boldsymbol{A}\boldsymbol{V}$ is known to be non-singular. These are;

- $\boldsymbol{A}$ is positive definite and $\mathcal{L} = \mathcal{K}$,
- $\boldsymbol{A}$ is not singular and $\mathcal{L} = \boldsymbol{A}\mathcal{K}$.

We refer the reader to [22] for proof and the analysis of the projection methods.

2.3.2 (Some) Krylov Subspace Methods

Krylov subspace methods are projection methods where subspace $\mathcal{K}$ is chosen to be a Krylov subspace. The Krylov subspace of dimension m is;

$$\mathcal{K}_m(A, r_0) = span\{\boldsymbol{r_0}, \boldsymbol{A}\boldsymbol{r_0}, \boldsymbol{A}^2\boldsymbol{r_0}, \ldots,, \boldsymbol{A}^{m-1}\boldsymbol{r_0}\},$$

where $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x_0}$. Different Krylov subspace methods are created depending on the choice of subspace $\mathcal{L}_m$ and preconditioning (which is the subject of subsection 2.3.3).

Notice that if $\mathcal{K} = \mathcal{K}_m$ approximation can be written as;

$$\boldsymbol{x}_m = \boldsymbol{x}_0 + \sum_{i=0}^{m-1} a_i A^i r_0,$$

or simply initial guess summed with linear combination of span of Krylov subspace. All Krylov subspace methods give this type of approximation, but choice of constraints (subspace $\mathcal{L}_m$) has great effect on iterative method. Two well-known choices are $\mathcal{L}_m = \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r_0})$ and $\mathcal{L}_m = \mathcal{K}_m(\boldsymbol{A}^T, \boldsymbol{r_0})$. Firstly we will discuss CG algorithm which assumes $\mathcal{L}_m = \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r_0})$, then later on we will discuss BICGSTAB which assumes $\mathcal{L}_m = \mathcal{K}_m(\boldsymbol{A}^T, \boldsymbol{r_0})$. The vectors in the basis of Krylov subspaces become almost linearly dependent for large m because of construction of basis, Krylov subspace methods involve orthogonalization schemes to prevent this phenomena.

First of these orthogonalization schemes is the Arnoldi's method [3]. It was first proposed to reduce a dense matrix to its Hessenberg form using unitary transformations. Later it was found to be efficient to approximate eigenvalues of large sparse matrices then it is extended to the solution of large sparse linear systems. Using Arnoldi's method, Full Orthogonalization Method (FOM) and its variations are developed. After these by using a similar idea to FOM, Generalized Minimum Residual Method (GMRES) is derived.

Another scheme is the symmetric Lanczos algorithm. It is in a way simplified version of Arnoldi's method when matrix is known to be symmetric. If matrix is symmetric then the Hessenberg form of matrix is symmetric tridiagonal. We will consider the Conjugate Gradient algorithm (CG) resulting from this algorithm in detail as CG is one of the methods we have used for solution of large sparse linear systems.

CG is one of the best known and widely used iterative techniques. It is also one of the very interesting as it can both be derived as a Krylov subspace method or as a solution of an optimization problem. Method is proven to be convergent for symmetric positive definite linear systems.

We will derive CG starting from Algorithm 1, but let us first introduce Lanczos's algorithm.

Given initial guess $\boldsymbol{x}_0$, Krylov subspace $\mathcal{K}_m$, an algorithm for solution of symmetric positive definite (SPD) linear systems is as follows;

Last step of the algorithm contains a tridiagonal solve to compute $\boldsymbol{y}_m = \boldsymbol{T}_m^{-1}(\beta\boldsymbol{e}_1)$. This can be included into the loop. First compute the LU factorization of $\boldsymbol{T}_m =$

```
Choose an initial vector $\boldsymbol{v}_1$ with $||\boldsymbol{v}_1||_2 = 1$. Set $\beta_1 \equiv 0$ and $v_0 \equiv 0$
for $j = 1, 2, \ldots, m$ do
    $\boldsymbol{w_j} := \boldsymbol{A}\boldsymbol{v_j} - \beta_j \boldsymbol{v_{j-1}}$
    $\alpha_j := (\boldsymbol{w_j}, \boldsymbol{v_j})$
    $\boldsymbol{w_j} := \boldsymbol{w_j} - \alpha_j \boldsymbol{v_j}$
    $\beta_{j+1} = ||\boldsymbol{w_j}||_2$. If $\beta_{j+1} = 0$ then Stop
    $\boldsymbol{v_{j+1}} := \boldsymbol{w_{j+1}}/\beta_{j+1}$
end
```

Algorithm 2: Lanczos's Algorithm

```
Compute $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, $\beta := ||r_0||_2$, $\boldsymbol{v}_1 := \boldsymbol{r}_0/\beta$
for $j = 1, 2, \ldots, m$ do
    $\boldsymbol{w_j} := \boldsymbol{A}\boldsymbol{v_j} - \beta_j \boldsymbol{v_{j-1}}$ (If $j = 1$, $\beta_1 \boldsymbol{v}_0 = 0$)
    $\alpha_j := (\boldsymbol{w_j}, \boldsymbol{v_j})$
    $\boldsymbol{w_j} := \boldsymbol{w_j} - \alpha_j \boldsymbol{v_j}$
    $\beta_{j+1} = ||\boldsymbol{w_j}||_2$. If $\beta_{j+1} = 0$ then Break Loop
    $\boldsymbol{v_{j+1}} := \boldsymbol{w_{j+1}}/\beta_{j+1}$
end
Set $\boldsymbol{T}_m = \text{tridiag}(\beta_i, \alpha_i, \beta_{i+1})$ and $\boldsymbol{V}_m = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m]$
Compute $\boldsymbol{y}_m = \boldsymbol{T}_m^{-1}(\beta \boldsymbol{e}_1)$ and $\boldsymbol{x}_m := \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{y}_m$
```

Algorithm 3: Lanczos's Algorithm for Linear Systems

$\boldsymbol{L}_m \boldsymbol{U}_m$. If $\boldsymbol{L}_m$ is assumed to be unit lower bidiagonal, then;

$$\boldsymbol{T}_m = \begin{pmatrix} 1 & & & \\ \lambda_2 & 1 & & \\ & \ddots & \ddots & \\ & & \lambda_m & 1 \end{pmatrix} \begin{pmatrix} \eta_1 & \beta_2 & & \\ & \eta_2 & \ddots & \\ & & \ddots & \beta_m \\ & & & \eta_m \end{pmatrix}.$$

The approximate solution is;

$$\begin{aligned} \boldsymbol{x}_m &= \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{y}_m = \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{T}_m^{-1}(\beta \boldsymbol{e}_1) \\ &= \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{U}_m^{-1} \boldsymbol{L}_m^{-1}(\beta \boldsymbol{e}_1). \end{aligned}$$

Defining $\boldsymbol{P}_m = \boldsymbol{V}_m \boldsymbol{U}_m^{-1}$ and $\boldsymbol{z}_m = \boldsymbol{L}_m^{-1}(\beta \boldsymbol{e}_1)$, it can simply be written as;

$$\boldsymbol{x}_m = \boldsymbol{x}_0 + \boldsymbol{P}_m \boldsymbol{z}_m.$$

Last column of $\boldsymbol{P}_m$ can be computed as;

$$\boldsymbol{p}_m = \eta_m^{-1}[\boldsymbol{v}_m - \beta_m \boldsymbol{p}_{m-1}]$$

where β_m is a scalar computed from Lanczos algorithm, η_m is computed from m-th step of Gaussian elimination process, and following are computed;

$$\lambda_m = \frac{\beta_m}{\eta_{m-1}} \tag{2.12}$$

$$\eta_m = \alpha_m - \lambda_m \beta_m. \tag{2.13}$$

Moreover, by defining $\xi_m = -\lambda_m \xi_{m-1}$ with $\xi_0 = ||\boldsymbol{r}_0||_2$, following can be shown;

$$\boldsymbol{z}_m = \begin{bmatrix} \boldsymbol{z}_{m-1} \\ \xi_m \end{bmatrix}.$$

Therefore, update of approximate solution at each step simply reduces to;

$$\boldsymbol{x}_m = \boldsymbol{x}_{m-1} + \xi_m \boldsymbol{p}_m.$$

Algorithm 4, which is called direct version of the Lanczos algorithm for linear systems (D-Lanczos), builds on this idea.

```
Compute $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, $\xi_1 := \beta := ||r_0||_2$, $\boldsymbol{v}_1 := \boldsymbol{r}_0/\beta$
Set $\lambda_1 = \beta_1 = 0$, $\boldsymbol{p}_0 = 0$
for $m = 1, 2, \ldots, M$ or until convergence do
    $\boldsymbol{w_m} := \boldsymbol{A v_m} - \beta_m \boldsymbol{v_{m-1}}$ amd $\alpha_m = (\boldsymbol{w}_j, \boldsymbol{v}_m)$
    If $m > 1$ then compute $\lambda_m = \frac{\beta_m}{\eta_{m-1}}$ and $\xi_m = -\lambda_m \xi_{m-1}$
    $\eta_m = \alpha_m - \lambda_m \beta_m$
    $\boldsymbol{p}_m = \eta_m^{-1}[\boldsymbol{v}_m - \beta_m \boldsymbol{p}_{m-1}]$
    $\boldsymbol{x}_m = \boldsymbol{x}_{m-1} + \xi_m \boldsymbol{p}_m$
    If $\boldsymbol{x}_m$ has converged then Stop
    $\boldsymbol{w_m} := \boldsymbol{w}_m - \alpha_m \boldsymbol{v_m}$
    $\beta_{m+1} = ||\boldsymbol{w}||_2, \boldsymbol{v}_{m+1} = \boldsymbol{w}/\beta_{m+1}$
end
```

Algorithm 4: D-Lanczos

We refer the reader to [22] for the proof and we give following properties of this algorithm;

Let $\boldsymbol{r}_m = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_m$ and $\boldsymbol{p}_m$, $m = 0, 1, 2, \ldots$, be as defined in Algorithm 4,

- $\boldsymbol{r}_m = \sigma_m \boldsymbol{v}_{m+1}$ for some σ_m, therefore residual vectors are orthogonal to each other.
- $\boldsymbol{p}_i$ vectors form an $\boldsymbol{A}$-conjugate set, i.e., $(\boldsymbol{A}\boldsymbol{p}_i, \boldsymbol{p}_j) = 0$ for $i \neq j$.

If this orthogonality and conjugacy properties are imposed as constraints, CG algorithm is obtained. The approximate solution can be written as;

$$\boldsymbol{x}_j = \boldsymbol{x}_{j-1} + \alpha_{j-1} \boldsymbol{p}_{j-1}. \tag{2.14}$$

Now consider

$$\boldsymbol{r}_j = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_j, \tag{2.15}$$

inserting 2.14 in 2.15, following is obtained;

$$\boldsymbol{r}_j = \boldsymbol{r}_{j-1} - \alpha_{j-1} \boldsymbol{A}\boldsymbol{p}_{j-1}. \tag{2.16}$$

By the orthogonality condition, $(\boldsymbol{r}_j, \boldsymbol{r}_{j-1}) = (\boldsymbol{r}_{j-1} - \alpha_{j-1}\boldsymbol{A}\boldsymbol{p}_{j-1}, \boldsymbol{r}_{j-1}) = 0$, therefore;

$$\alpha_{j-1} = \frac{(\boldsymbol{r}_{j-1}, \boldsymbol{r}_{j-1})}{(\boldsymbol{A}\boldsymbol{p}_{j-1}, \boldsymbol{r}_{j-1})}. \tag{2.17}$$

$\boldsymbol{p}_{j+1}$ can be determined as a linear combination of $\boldsymbol{r}_{j+1}$ and $\boldsymbol{p}_j$,

$$\boldsymbol{p}_{j+1} = \boldsymbol{r}_{j+1} + \beta_j \boldsymbol{p}_j. \tag{2.18}$$

Using A-conjugacy condition,

$$(\boldsymbol{A}\boldsymbol{p}_i, \boldsymbol{r}_i) = (\boldsymbol{A}\boldsymbol{p}_i, \boldsymbol{p}_i - \beta_{i-1}\boldsymbol{p}_{i-1}) = (\boldsymbol{A}\boldsymbol{p}_i, \boldsymbol{p}_i) - (\boldsymbol{A}\boldsymbol{p}_i, \beta_{i-1}\boldsymbol{p}_{i-1}) = (\boldsymbol{A}\boldsymbol{p}_i, \boldsymbol{p}_i), \tag{2.19}$$

and as $(\boldsymbol{A}\boldsymbol{p}_i, \boldsymbol{p}_{i+1}) = 0$ and considering 2.18;

$$\beta_j = -\frac{(\boldsymbol{r}_{j+1}, \boldsymbol{A}\boldsymbol{p}_j)}{(\boldsymbol{p}_j, \boldsymbol{A}\boldsymbol{p}_j)}. \tag{2.20}$$

Using 2.15 further simplication can be done, resulting;

$$\beta_j = -\frac{(\boldsymbol{r}_{j+1}, \boldsymbol{r}_{j+1})}{(\boldsymbol{r}_j, \boldsymbol{r}_j)}. \tag{2.21}$$

Also as a result of 2.19, $\alpha_{j-1} = (\boldsymbol{r}_{j-1}, \boldsymbol{r}_{j-1})/(\boldsymbol{A}\boldsymbol{p}_{j-1}, \boldsymbol{p}_{j-1})$. Using above calculations, we will give algorithm for CG method(Algorithm 5).

```
Compute $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$ and set $\boldsymbol{p}_0 = \boldsymbol{r_0}$
for $j = 1, 2, \ldots, M$ or until convergence do
    $\alpha_{j-1} = \frac{(\boldsymbol{r}_{j-1}, \boldsymbol{r}_{j-1})}{(\boldsymbol{A}\boldsymbol{p}_{j-1}, \boldsymbol{p}_{j-1})}$
    $\boldsymbol{x}_j = \boldsymbol{x}_{j-1} + \alpha_{j-1}\boldsymbol{p}_{j-1}$
    $\boldsymbol{r}_j = \boldsymbol{r}_{j-1} - \alpha_{j-1}\boldsymbol{A}\boldsymbol{p}_{j-1}$
    $\beta_{j-1} = -\frac{(\boldsymbol{r}_j, \boldsymbol{r}_j)}{(\boldsymbol{r}_{j-1}, \boldsymbol{r}_{j-1})}$
    $\boldsymbol{p}_j = \boldsymbol{r}_j + \beta_{j-1}\boldsymbol{p}_{j-1}$
end
```

Algorithm 5: Conjugate Gradient

If matrices are symmetric and positive definite, assuming exact arithmetic, CG is guaranteed to converge. Situation is different in case of finite precision arithmetic, as round-off errors may and will accumulate and prevent finding the exact solution. But usually if coefficient matrix $\boldsymbol{A}$ is well conditioned, with each iteration approximate solution progressively gets better. Usually CG iterations are stopped when $||\boldsymbol{r}_m||/||\boldsymbol{r}_0||$ drops below a predefined small tolerance, like 10^{-8}. Due to numerical errors that may occur, such tolerance may not be achievable. Therefore, a maximum number of iterations is set to prevent infinite loops (denoted as M in algorithms).

It is beneficial to consider number of operations per iteration. There are 4 inner products, 3 axpy operations ($\boldsymbol{y} \leftarrow \alpha\boldsymbol{x} + \boldsymbol{y}$) and 2 matrix-vector products. These can be reduced to 3 inner products, 3 axpy operations and 1 matrix-vector product. Such an algorithm is presented in Algorithm 6.

```
Compute $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$ and set $\boldsymbol{p}_0 = \boldsymbol{r_0}$
for $j = 1, 2, \ldots, M$ or until convergence do
    $rr = (\boldsymbol{r}_{j-1}, \boldsymbol{r}_{j-1})$
    $ap = \boldsymbol{A}\boldsymbol{p}_{j-1}$
    $\alpha_{j-1} = \frac{rr}{(ap, \boldsymbol{p}_{j-1})}$
    $\boldsymbol{x}_j = \boldsymbol{x}_{j-1} + \alpha_{j-1}\boldsymbol{p}_{j-1}$
    $\boldsymbol{r}_j = \boldsymbol{r}_{j-1} - \alpha_{j-1} ap$
    $\beta_{j-1} = -\frac{(\boldsymbol{r}_j, \boldsymbol{r}_j)}{rr}$
    $\boldsymbol{p}_j = \boldsymbol{r}_j + \beta_{j-1}\boldsymbol{p}_{j-1}$
end
```

Algorithm 6: A more cost-efficient Conjugate Gradient

For general matrices, there exists an extension of CG which makes use of Lanczos Biorthogonalization algorithm, which is called Bi-Conjugate Gradient Stabilized (BiCGStab). BiCGStab have quite interesting properties but it is hard to make a theoretical analysis. BiCGSTAB is not guaranteed to converge and there is no convergence proof. For details we refer the reader to [26], general BiCGStab algorithm is in Algorithm 7.

```
Compute $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, $\boldsymbol{r}_0^*$ arbitrary, $\boldsymbol{p}_0 = \boldsymbol{r_0}$;
for $j = 1, 2, \ldots, M$ or until convergence do
    $\alpha_{j-1} = \frac{(\boldsymbol{r}_{j-1}, \boldsymbol{r}_0^*)}{(\boldsymbol{A}\boldsymbol{p}_{j-1}, \boldsymbol{r}_0^*)}$
    $\boldsymbol{s}_{j-1} = \boldsymbol{r}_{j-1} - \alpha_{j-1}\boldsymbol{A}\boldsymbol{p}_{j-1}$
    $\omega_{j-1} = (\boldsymbol{A}\boldsymbol{s}_{j-1}, \boldsymbol{s}_{j-1})/(\boldsymbol{A}\boldsymbol{s}_{j-1}, \boldsymbol{A}\boldsymbol{s}_{j-1})$
    $\boldsymbol{x}_j = \boldsymbol{x}_{j-1} + \alpha_{j-1}\boldsymbol{p}_{j-1} + \omega_{j-1}\boldsymbol{s}_{j-1}$
    $\boldsymbol{r}_j = \boldsymbol{s}_{j-1} - \omega_{j-1}\boldsymbol{A}\boldsymbol{s}_{j-1}$
    $\beta_{j-1} = -\frac{(\boldsymbol{r}_j, \boldsymbol{r}_0^*)}{(\boldsymbol{r}_{j-1}, \boldsymbol{r}_j^0)} \times \frac{\alpha_{j-1}}{\omega_{j-1}}$
    $\boldsymbol{p}_j = \boldsymbol{r}_j + \beta_{j-1}(\boldsymbol{p}_{j-1} - \omega_{j-1}\boldsymbol{A}\boldsymbol{p}_{j-1})$
end
```

Algorithm 7: Bi-Conjugate Gradient Stabilized

Similar to CG case let us consider number operations per iteration for BiCGStab. There are 6 inner products, 6 axpy operations and 7 matrix-vector products. But using the idea in CG these can be reduced to 6 inner products, 6 axpy operations and 2 matrix-vector products.

This show us if we can use both CG and BiCGStab to solve a linear system, it is better to use CG as it is known to converge and number of operations per iteration is much less than BiCGStab. Nevertheless, in the cases where CG can not be used, we have used BiCGStab as it is a good alternative.

```
Compute $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, $\boldsymbol{r}_0^*$ arbitrary, $\boldsymbol{p}_0 = \boldsymbol{r_0}$;
for $j = 1, 2, \ldots, M$ or until convergence do
    $ap = \boldsymbol{A}\boldsymbol{p}_{j-1}$
    $as = \boldsymbol{A}\boldsymbol{s}_{j-1}$
    $\alpha_{j-1} = \frac{(\boldsymbol{r}_{j-1}, \boldsymbol{r}_0^*)}{(ap, \boldsymbol{r}_0^*)}$
    $\boldsymbol{s}_{j-1} = \boldsymbol{r}_{j-1} - \alpha_{j-1} ap$
    $\omega_{j-1} = (as, \boldsymbol{s}_{j-1})/(as, as)$
    $\boldsymbol{x}_j = \boldsymbol{x}_{j-1} + \alpha_{j-1}\boldsymbol{p}_{j-1} + \omega_{j-1}\boldsymbol{s}_{j-1}$
    $\boldsymbol{r}_j = \boldsymbol{s}_{j-1} - \omega_{j-1} as$
    $\beta_{j-1} = -\frac{(\boldsymbol{r}_j, \boldsymbol{r}_0^*)}{(\boldsymbol{r}_{j-1}, \boldsymbol{r}_j^0)} \times \frac{\alpha_{j-1}}{\omega_{j-1}}$
    $\boldsymbol{p}_j = \boldsymbol{r}_j + \beta_{j-1}(\boldsymbol{p}_{j-1} - \omega_{j-1} ap)$
end
```

Algorithm 8: A more cost efficient Bi-Conjugate Gradient Stabilized

2.3.3 Preconditioners

Iterative methods are not robust when compared to direct solvers. Therefore, even though they are memory and computation efficient, in some cases (especially for large sparse linear systems) direct solvers are preferred over iterative methods. Preconditioning improves both robustness and efficiency of the iterative methods. Preconditioning is basically transforming a linear system of equations into another one with same solution but has better eigenvalue distribution, hence easier to solve using iterative methods.

First we want to discuss what makes a preconditioner a good preconditioner. Let $\boldsymbol{M}$ be preconditioner. First requirement is that the computation of $\boldsymbol{M}$. It should not take a lot of time so that choosing iterative methods over direct methods stays feasible, even in presence of computational overhead preconditioner introduces. Most important requirement is that application of preconditioner must be computationally inexpensive. All Krylov subspace methods apply preconditioner at each iteration, because this approach is usually cheaper and flexible rather than applying before starting to solve linear system and change the system. Preconditioner should be nonsingular since its application is most of the time done by solving $\boldsymbol{M}\boldsymbol{x} = \boldsymbol{b}$. Also $\boldsymbol{M}$ should approximate the original coefficient matrix in a way which will be elaborated later.

There are three ways a preconditioner can be applied. If it is applied from left, called left preconditioning,

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}, \tag{2.22}$$

if it is applied from right, called right preconditioning,

$$\boldsymbol{A}\boldsymbol{M}^{-1}\boldsymbol{M}\boldsymbol{x} = \boldsymbol{A}\boldsymbol{M}^{-1}\boldsymbol{u} = \boldsymbol{b}. \tag{2.23}$$

Note that, $\boldsymbol{u} = \boldsymbol{M}\boldsymbol{x}$. Therefore to find solution $\boldsymbol{x}$ another linear system must be solved. Thirdly, if preconditioner is available in the factored form, left and right preconditioning can be used at same time, called split preconditioning. Let $\boldsymbol{M} = \boldsymbol{M}_L\boldsymbol{M}_R$, it can be applied in following way,

$$\boldsymbol{M}_L^{-1}\boldsymbol{A}\boldsymbol{M}_R^{-1}\boldsymbol{M}_R\boldsymbol{x} = \boldsymbol{M}_L^{-1}\boldsymbol{A}\boldsymbol{M}_R^{-1}\boldsymbol{u} = \boldsymbol{M}_L^{-1}\boldsymbol{b} \tag{2.24}$$

with $\boldsymbol{u} = \boldsymbol{M}_R\boldsymbol{x}$.

If the original matrix is symmetric, to be able to use methods that work for symmetric matrices, it seems like preconditioner should be a carefully chosen split preconditioner. But this is not true as there are methods to preserve symmetry even when preconditioner is not available in factored form. We will consider preconditioned version of CG (PCG) to explain how left preconditioning may be applied without disturbing symmetry.

Assume we are trying to solve $\boldsymbol{Ax} = \boldsymbol{b}$ and we have preconditioner $\boldsymbol{M}$. Necessarily $\boldsymbol{A}$ and $\boldsymbol{M}$ must be SPD. Then $\boldsymbol{M}$ can be applied in three ways (left, right or split). If left or right preconditioning is chosen, preconditioned system may not be symmetric (because $(\boldsymbol{AB})^T = \boldsymbol{B}^T\boldsymbol{A}^T = \boldsymbol{BA} \neq \boldsymbol{AB}$ for some symmetric matrices $\boldsymbol{A}$ and $\boldsymbol{B}$). In last case since $\boldsymbol{M}$ is SPD, it can be Cholesky factorized, i.e. there exists lower triangular matrix $\boldsymbol{L}$ such that $\boldsymbol{M} = \boldsymbol{LL}^T$. So split preconditioning preserves SPD property of original linear system.

Often the Cholesky factorization of the preconditioner is not available and hard to compute from the preconditioner itself. Even so, it is possible to preserve symmetry using left or right preconditioning. Consider;

$$(\boldsymbol{M}^{-1}\boldsymbol{Ax}, \boldsymbol{y})_{\boldsymbol{M}} = (\boldsymbol{Ax}, \boldsymbol{y}) = (\boldsymbol{x}, \boldsymbol{Ay}) = (\boldsymbol{x}, \boldsymbol{MM}^{-1}\boldsymbol{Ay}) = (\boldsymbol{x}, \boldsymbol{M}^{-1}\boldsymbol{Ay})_{\boldsymbol{M}}$$

which shows $\boldsymbol{M}^{-1}\boldsymbol{A}$ is self-adjoint for the M-inner product. Thus, using M-inner product rather that usual Euclidean inner product preserves symmetry.

Introducing this inner product to CG, left preconditioned Conjugate Gradient method is obtained. We will now derive PCG. Let $\boldsymbol{r}_j = \boldsymbol{b} - \boldsymbol{Ax}_j$ and $\boldsymbol{z}_j = \boldsymbol{M}^{-1}\boldsymbol{r}_j$, residual of the preconditioned system. Then

$$\begin{aligned}
&\alpha_j = \frac{(\boldsymbol{z}_j, \boldsymbol{z}_j)_{\boldsymbol{M}}}{(\boldsymbol{M}^{-1}\boldsymbol{Ap}_j, \boldsymbol{p}_j)_{\boldsymbol{M}}} = \frac{(\boldsymbol{r}_j, \boldsymbol{z}_j)}{(\boldsymbol{Ap}_j, \boldsymbol{p}_j)},\\
&\boldsymbol{x}_j = \boldsymbol{x}_{j-1} + \alpha_{j-1}\boldsymbol{p}_{j-1},\\
&\boldsymbol{r}_j = \boldsymbol{r}_{j-1} - \alpha_{j-1}\boldsymbol{Ap}_{j-1} \text{ and } \boldsymbol{z}_j = \boldsymbol{M}^{-1}\boldsymbol{r}_j,\\
&\beta_j = -\frac{(\boldsymbol{z}_{j+1}, \boldsymbol{z}_{j+1})_{\boldsymbol{M}}}{(\boldsymbol{z}_j, \boldsymbol{z}_j)_{\boldsymbol{M}}} = -\frac{(\boldsymbol{r}_{j+1}, \boldsymbol{z}_{j+1})}{(\boldsymbol{r}_j, \boldsymbol{z}_j)},\\
&\boldsymbol{p}_{j+1} = \boldsymbol{z}_{j+1} + \beta_j\boldsymbol{p}_j.
\end{aligned}$$

Algorithm 9 summarizes this calculations into the algorithm of left preconditioned CG. Split preconditioned CG is a direct consequence of left preconditioned with some more auxillary vectors introduced and iterations are identical in exact arithmetic. Right preconditioned CG can be derived in a similar way to left preconditioned CG by observing that $\boldsymbol{AM}^{-1}$ is self-adjoint for the $\boldsymbol{M}^{-1}$-inner product.

It is beneficial to discuss preconditioners and preconditioning techniques in detail. It was mentioned that idea of using a preconditioner is to transform the linear systems to linear systems which are easier to solve with iterative solvers. Convergence of iterative solvers strongly depends on the eigenvalues hence on the condition number of the

```
Compute $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$ and set $\boldsymbol{p}_0 = \boldsymbol{r_0}$
for $j = 0, 1, 2, \ldots, M$ or until convergence do
    $\alpha_j = \frac{(\boldsymbol{r}_j, \boldsymbol{z}_j)}{(\boldsymbol{A}\boldsymbol{p}_j, \boldsymbol{p}_j)}$
    $\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \alpha_j \boldsymbol{p}_j$
    $\boldsymbol{r}_{j+1} = \boldsymbol{r}_j - \alpha_j \boldsymbol{A}\boldsymbol{p}_j$
    $\boldsymbol{z}_{j+1} = \boldsymbol{M}^{-1}\boldsymbol{r}_j$
    $\beta_j = -\frac{(\boldsymbol{r}_{j+1}, \boldsymbol{z}_{j+1})}{(\boldsymbol{r}_j, \boldsymbol{z}_j)}$
    $\boldsymbol{p}_{j+1} = \boldsymbol{z}_{j+1} + \beta_j \boldsymbol{p}_j$
end
```

Algorithm 9: Left Preconditioned Conjugate Gradient

coefficient matrix. Consider $\boldsymbol{Ax} = \boldsymbol{b}$ with $cond(\boldsymbol{A}) >> 1$. Assume $\boldsymbol{M}$ is a left preconditioner to the system, then we expect that $cond(\boldsymbol{A}) > cond(\boldsymbol{M}^{-1}\boldsymbol{A}) \geq 1$. Earlier it was told that $\boldsymbol{M}$ should resemble coefficient matrix $\boldsymbol{A}$ in a way. It is because, if it was not too expensive to compute, the best preconditioner for the linear system would the coefficient matrix of linear system since $cond(\boldsymbol{A}^{-1}\boldsymbol{A}) = 1$. But this requires computation of inverse of $\boldsymbol{A}$, which is computationally expensive (as expensive as solving directly, therefore meaningless). Cheapest preconditioner in that sense is identity matrix, because it is free, but does not affect the condition number of the linear system at all. So a compromise must be done. We will consider three of those compromises, which are Incomplete LU factorization (ILU), Block-Jacobi and sparse approximate inverse (SAI) type preconditioners, but we refer the reader to the surveys [4] and [9] for a more detailed discussion of various preconditioning techniques.

The main idea in ILU-type preconditioners is to find an approximate LU factorization of the coefficient matrix $\boldsymbol{A}$ where $\boldsymbol{M} = \widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{U}} \approx \boldsymbol{A}$. Notice that this kind of preconditioning is appropriate for split preconditioning and if $\widetilde{\boldsymbol{U}}$ is chosen such that $\widetilde{\boldsymbol{U}} = \widetilde{\boldsymbol{L}}^T$, called Incomplete Cholesky Factorization (ICC), application of preconditioner preserves symmetry by default. We impose the condition of that $\widetilde{\boldsymbol{L}}$ and $\widetilde{\boldsymbol{U}}$ must be sparse. Satisfying this condition is relatively easy. A nonzero structure may be assumed in their construction or by dropping elements which are less than some threshold in absolute value.

An example for first case is to choose nonzero structure as nonzero structure of $\boldsymbol{A}$, this is called zero fill-in ILU (ILU(0)). In general nonzero structure can be assumed of nonzero structure of $\boldsymbol{A}^{(p+1)}$ for $p \geq 0$, called ILU(p). Assume S is nonzero structure of $\boldsymbol{A}$, an algorithm can be given as follows;

This algorithm is an in-place algorithm, which saves ILU factors over coefficient matrix $\boldsymbol{A}$. $\widetilde{\boldsymbol{L}}$ assumed to be lower unit triangular matrix, so strictly lower part of $\boldsymbol{A}$ summed with identity matrix of suitable size is $\widetilde{\boldsymbol{L}}$ while $\widetilde{\boldsymbol{U}}$ is the upper part of $\boldsymbol{A}$ after algorithm ends. Note that, this algorithm fails if $a_{kk} = 0$ so another algorithm which employs pivoting must be used in those cases. ILU has many implementations but a good example is Euclid [18], which is a parallel ILU type preconditioner.

```
for $i = 0, 1, 2, \ldots, n$ do
    for $k = 0, 1, 2, \ldots, i-1$ and for $(i, k) \in S$ do
        Compute $a_{ik} = a_{ik}/a_{kk}$
        for $j = k+1, 2, \ldots, n$ do
            Compute $a_{ij} = a_{ij} - a_{ik}a_{kj}$
        end
    end
end
```

Algorithm 10: ILU(0)

Block-Jacobi preconditioner idea is simpler than ILU idea. Considering $\boldsymbol{Ax} = \boldsymbol{b}$ with;

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & & \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} & \boldsymbol{A}_{23} & \\ & \boldsymbol{A}_{32} & \boldsymbol{A}_{33} & \boldsymbol{A}_{34} \\ & & \boldsymbol{A}_{43} & \boldsymbol{A}_{44} \end{pmatrix},$$

where $\boldsymbol{A}_{ij}$'s are square submatrices (or blocks) of $\boldsymbol{A}$. Some i, j values are skipped to emphasize sparsity but structure of coefficient matrix is only given as an example. Before starting to construct Block-Jacobi preconditioner, we must decide how many blocks will be used. If one block is to be used as preconditioner then $\boldsymbol{M} = \boldsymbol{A}$. In this case for application of preconditioner, another method should be used to decrease the cost of application. An example may be using ILU to apply the preconditioner. If more blocks are to be used, for example two and four blocks;

$$\boldsymbol{M_2} = \begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & & \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} & & \\ & & \boldsymbol{A}_{33} & \boldsymbol{A}_{34} \\ & & \boldsymbol{A}_{43} & \boldsymbol{A}_{44} \end{pmatrix}$$

and

$$\boldsymbol{M_4} = \begin{pmatrix} \boldsymbol{A}_{11} & & & \\ & \boldsymbol{A}_{22} & & \\ & & \boldsymbol{A}_{33} & \\ & & & \boldsymbol{A}_{44} \end{pmatrix},$$

application of preconditioner may be formulized as independent direct solution of k much smaller linear systems (compared to $\boldsymbol{A}$), where k is the number of blocks. Even more blocks may be used with the limiting case of number of blocks being the dimension of coefficient matrix. Furthermore, sparse DS factorization based schemes could be considered as a generalization of the block-jacobi including important nonzeros in the off-diagonal blocks [20].

SAI idea is, given a linear system $\boldsymbol{Ax} = \boldsymbol{b}$, to find a sparse matrix $\boldsymbol{M}$ such that the Frobenius norm of the error $||\boldsymbol{MA} - \boldsymbol{I}||_F < \epsilon$ for some $\epsilon > 0$ under a sparsity constraint on $\boldsymbol{M}$. If the structure of $\boldsymbol{A}^{-1}$ is known, it can be used for the sparsity pattern of $\boldsymbol{M}$. If it is not known, one of common approaches is to assume that preconditioner $\boldsymbol{M}$ has the same nonzero structure as $\boldsymbol{A}^k$ for some k, but as k gets larger this is costly, and there is a limit to ϵ as the structure is fixed. Alternatively, for a given ϵ, trying to find a structure for M that satisfies given ϵ is another possibility. Well-known examples are ParaSails [12] for assuming a priori structure and SPAI [16] which tries to find an appropriate structure to satisfy ϵ condition.

ILU type preconditioners are not scalable as much as other two due to inherently sequential nature and limited scalability of triangular solves. Also computation of preconditioner has limited scalability.

Block-Jacobi preconditioners are quite parallel due to application of preconditioner can be done independently for each block. But as number of blocks increase, preconditioner starts to diverge from coefficient matrix, which may cause in false or slow convergence.

SAI type of preconditioners are expected to be scalable on parallel computing platforms since computing the preconditioner matrix can be split into completely independent linear least squares problems. Another reason is that applying the preconditioner is just a matrix-vector multiplication which is usually possible to parallelize.

In next section, parallelization of iterative solvers and preconditioners will be discussed.

2.4 Parallelization

Advancement and large availability of parallel computing platforms in last decade, put parallelization of existing algorithms to an important place for solution of large scale problems. As three dimensional models have gained importance in applications, related problem of solving large sparse linear systems gained importance too which are quite large scale problems.

Solution methods for this problems have different advantages and disadvantages. Direct methods are robust, but they require a lot of storage and scalability is limited. Most popular and known parallel direct solvers are Pardiso [23], MUMPS [2], WSMP [17] and SuperLU [19]. Iterative solvers require much less memory and far more scalable and easier to implement, but they are not robust. Preconditioned iterative methods are more efficient and robust, yet with more cost. Packages like PETSc [5], Hypre [15] provide both parallelized versions of iterative solvers and preconditioners.

Discussion of parallel computing platforms is beneficial before further inspection of parallelization. There are two types of parallel computing platforms which are used and compared in this thesis, even though there are more models. These two are shared memory architectures and distributed memory architectures.

Shared memory architectures have many cores connected to a single (usually large) memory unit that is addressable by any core. Common bottlenecks in these kind of architectures are cache coherence and memory bandwidth. Memory conflicts, actions those disrupt cache coherence, happen when a core tries to change a data which other cores need, or two or more cores tries to write to same place, problem is more difficult when multi levels of caches are used. Memory bandwidth is more related to reading data from memory. If an algorithm requires frequent reads of large amount of data from memory, computation step of algorithm waits for reading to complete. But if at any moment data that must be read is much larger than bandwidth (data that can be read in unit time), performance of algorithm will suffer.

A distributed memory architecture is a collection of processors (each having their "private" memory storages) which are connected through a high speed network. In this model, processors can not read any data that is not stored in their memory. As data is not shared, if any data exchange is required that must be foreseen and programmed by programmer. Advantage of this architecture is that the aggregated memory bandwidth is quite large and is no longer a bottleneck. A disadvantage is the need for message passing. Network communications, even though may be quite fast, are not as fast as memory reads and involves higher latency. Usually it is possible to minimize communication by manipulating data, but for many problems communication is inherently required.

There are many different approaches to parallelize direct methods on shared memory architectures but relatively simpler than distributed memory architectures. Consider Gaussian Elimination without pivoting or iterative refinement. All cores can read the first line of coefficient matrix and make relative operations at the same time. Each core should operate on distinct lines to prevent memory conflicts. Same algorithm can be used on distributed memory structures, but at each step of Gaussian Elimination a vector is broadcasted and each different processors does elimination on their local rows. This have two problems, (1) a global communication (i.e. with all processing units on the structure) is required at each step and (2) some processors may complete their job due to smaller number of rows that should be eliminated and wait for others to finish. Solution for these problems is complicated and out of scope of this thesis. For further details, we refer the reader to [23], [2],[17] and [19].

Parallelization of a Krylov subspace method involves parallelization of each step that the algorithm involves. All Krylov subspace methods uses same three operations, namely, inner product, summation of two vectors and matrix-vector multiplication. If methods are also preconditioned, depending on type of preconditioner, application of the preconditioner is also another operation that is necessary. Therefore, basically by parallelizing these operations, Krylov subspace methods can be parallelized. Let us to divide and conquer each of operations, except linear system solve as we have already considered. Firstly, inner product of two column vectors is defined as;

$$(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^T \boldsymbol{y} = x_1 y_1 + x_2 y_2 + x_3 y_3 + \cdots + x_m y_m + \cdots + x_n y_n.$$

This can be written as,

$$(\boldsymbol{x}, \boldsymbol{y}) = (x_1 y_1 + \cdots + x_m y_m) + (x_{m+1} y_{m+1} + \cdots + x_n y_n) = (\boldsymbol{x}_1, \boldsymbol{y}_1) + (\boldsymbol{x}_2, \boldsymbol{y}_2),$$

where $\boldsymbol{x}_1 = [x_1, \ldots, x_m]$ and $\boldsymbol{x}_2 = [x_{m+1}, \ldots, x_n]$, similarly for $\boldsymbol{y}$. As it can be clearly seen inner product is an embarrasingly parallel operation. If a distributed memory architecture is used, processing units do not have to send-receive data except at the last part where individual inner products must be summed. Summation of two vectors is quite similar to inner product. Only difference is that, if result should be gathered in one core on distributed memory structures then rather than sending numbers whole vectors (arrays) must be send using a parallel reduce operation. Therefore summation of two vectors may also be considered as embarrasingly parallel.

Matrix-vector product requires special attention, even though it can simply be thought as multiple inner products. This is a valid approach to problem and a simple algorithm arises from this idea. Attention must be paid on the part concerning the storage of sparse matrix. Sparse matrices are not stored as two dimensional arrays, instead they are usually stored in multiple one dimentional array using one of the many formats. Let us consider the matrix 2.1 and its more common CSR representation,(— shows where data is divide between processors).

```
IA = [1 3 | 5 7 8 9]
JA = [1 3 2 4 | 1 3 2 4 5]
VA = [1 2 3 4 | 9 5 8 6 7]
```

Assume, it is necessary to multiply this matrix with a vector of ones (thus practically to find row sums). Firstly second core should send `IA(3)`=5 to first processor as it needs that information to know where to stop. We should also make another assumption for distribution of ones vector. It usually is not probable to hold full vector in each individual memory for large scale problems. So let's assume first two elements of ones is in first processor and rest is in second processor.

$$\boldsymbol{A1} = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 3 & 0 & 4 & 0 \\ \hline 9 & 0 & 5 & 0 & 0 \\ 0 & 8 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 7 \end{pmatrix} \begin{bmatrix} 1 \\ 1 \\ \hline 1 \\ 1 \\ 1 \end{bmatrix}$$

Note that, if structure is a distributed memory structure processors must communicate for the parts of ones vector they do not have. But before that we will introduce algorithm for CSR matrix vector multiplication. As long as processors have all the

```
for $i=1,2,\dots,n+1$ do
    for $j=IA(i),\dots,IA(i+1)$ do
        y(i) = y(i) + VA(j)*x(JA(j))
    end
end
```

Algorithm 11: CSR Matrix-Vector Multiplication

information they need, they can run this algorithm independently and gather the resulting vector on a single core upon conclusion, if necessary. But for example second processor needs first element of ones vector to complete multiplication but does not have. So a parallel algorithm for CSR MV is;

Processors should not wait for confirmation from other processor as this algorithm is run on all processors at the same time, all will be waiting for confirmation but since

```
for i=local_start,...,local_end+1 do
    Determine out of processor data needed,
    Send information about necessary data to relative processor, do not wait for
    confirmation of receiving,
    Before starting to wait data from other processors, check if they need any
    data from local memory,
    Send any data asked, do not wait for confirmation of receiving
    Receive data needed,
    for j=IA(i),...,IA(i+1) do
        y(i) = y(i) + VA(j)*x(JA(j))
    end
end
```

Algorithm 12: CSR Matrix-Vector Multiplication

none of them is received anything they will not send confirmations. This will cause a deadlock. Send-receive operations that does not wait for confirmations are called immediate (unblocking) send-receive operations.

If method is preconditioned, depending on preconditioner, application of preconditioner is either linear system solve (in case of ILU and Block-Jacobi) or matrix-vector multiplication (SAI). As both are covered, let us discuss parallelization of computation of preconditioners.

Computation of Block-Jacobi is straight forward if number of blocks is chosen to be number of cores, which is the usual approach. For example partition for matrix 2.1, with 3 cores, is;

$$\boldsymbol{A} = \left(\begin{array}{ccccc} 1 & 0 & 2 & 0 & 0 \\ 0 & 3 & 0 & 4 & 0 \\ \hline 9 & 0 & 5 & 0 & 0 \\ 0 & 8 & 0 & 6 & 0 \\ \hline 0 & 0 & 0 & 0 & 7 \end{array}\right).$$

Each core now only should keep the part on the diagonal and ignore rest for preconditioner, therefore;

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{M}_1 \\ \boldsymbol{M}_2 \\ \boldsymbol{M}_3 \end{pmatrix},$$

where

$$\boldsymbol{M}_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \end{pmatrix}, \boldsymbol{M}_2 = \begin{pmatrix} 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \end{pmatrix}, \boldsymbol{M}_3 = \begin{pmatrix} 0 & 0 & 0 & 0 & 7 \end{pmatrix}.$$

This operation can be done without any communication. Computation of Block-Jacobi preconditioners are quite parallel. Application of preconditioner is also can be thought as independent solutions of, in this case, three linear systems of equations. The problem with this type preconditioner is that as number of blocks increase, quality of the preconditioner degrades. It starts to diverge from coefficient matrix. With the increasing number of iterations, the scalability of these kind of preconditioner are usually limited.

ILU-type preconditioners are inherently sequential in their calculation. Similar to direct solution methods for linear systems (as ILU idea takes root from complete LU factorization), at each step a vector can be broadcasted and each processor may act upon that information. Another possibility is to find Block-Jacobi preconditioner with blocks overlapping to a degree and find ILU factorization of those blocks and use as a preconditioner for coefficient matrix. More complicated parallelizations of ILU preconditioners may be found in [22] and [18].

In contrast to ILU-type preconditioners, computation of sparse approximate inverse (SAI) type preconditioners is parallel. The idea is to solve the following optimization problem under some constraints to the nonzero structure, which we mentioned earlier, of $\boldsymbol{M}^{-1}$;

$$\min ||\boldsymbol{M}^{-1}\boldsymbol{A} - \boldsymbol{I}||. \tag{2.25}$$

Rather than solving that problem, by applying triangle inequality, we can solve;

$$\min \left(||\boldsymbol{M}_1^{-1}\boldsymbol{A} - \boldsymbol{I}^1|| + ||\boldsymbol{M}_2^{-1}\boldsymbol{A} - \boldsymbol{I}^2|| + \cdots + ||\boldsymbol{M}_p^{-1}\boldsymbol{A} - \boldsymbol{I}^p|| \right), \tag{2.26}$$

where p is number of workers, $\boldsymbol{M}_n^{-1}$ denotes nth part of $\boldsymbol{M}^{-1}$ when it is partitioned rowwise and $\boldsymbol{I}^n$ is corresponding part when $\boldsymbol{I}$ is partitioned columnwise. Now, it is obvious that;

$$\min \left(||\boldsymbol{M}_1^{-1}\boldsymbol{A} - \boldsymbol{I}^1|| + ||\boldsymbol{M}_2^{-1}\boldsymbol{A} - \boldsymbol{I}^2|| + \cdots + ||\boldsymbol{M}_p^{-1}\boldsymbol{A} - \boldsymbol{I}^p|| \right) = \tag{2.27}$$

$$\min ||\boldsymbol{M}_1^{-1}\boldsymbol{A} - \boldsymbol{I}^1|| + \min ||\boldsymbol{M}_2^{-1}\boldsymbol{A} - \boldsymbol{I}^2|| + \cdots + \min ||\boldsymbol{M}_p^{-1}\boldsymbol{A} - \boldsymbol{I}^p||. \tag{2.28}$$

Then each minimization problem can be solved by a different processor independently from others.

CHAPTER 3

APPLICATIONS

In this chapter, test problems are introduced. These problems are important in sense of industrial applications and scientific interest, which are explained in detail in respective sections. Their constructions are illustrated, even though not in great detail, for sake of emphasising the challenges those arise in solution of resulting linear system of equations.

Results and figures in this chapter are (accepted) to be published in [1, 24].

3.1 TOPOLOGY OPTIMIZATION

Topology optimization, also sometimes called as layout optimization, is a multidisciplinary design tool with purpose of minimizing usage of precious materials and optimizing material distribution. This is of quite importance by itself as resources on earth are finite and, in case of industrial applications, are expensive. Using right amount of materials in right way benefits both the environment and the industry.

Challenge in these problems are threefold. Successive solution of linear systems of equations is necessary to find solution to optimization problem. This is the most time consuming part especially for large, three dimensional problems. For practical three dimensional problems, number of unknowns can easily reach to millions. Lastly, the condition number of linear systems worsens as the optimization progresses.

3.1.1 Background

Bendsoe and Kikuchi in 1988 [7] applied the density-based material distribution initially to structural problems. Method is later applied to heat transfer and fluid flows [10, 11, 13]. Optimization steps in this method consists discretization of linear elasticity equilibrium equations with finite element method, solution of resulting linear system of equations and removing or adding material, via help of the well-known optimality criteria method [8] with gradient-based sensitivity calculations, to an initial design domain until an optimal distribution of material for a given volume ratio is achieved. At each optimization step, condition number of coefficient matrix of linear systems of equations worsens to the point of becoming singular and becomes highly stiff for iterative solvers. Therefore, achieving good scalability when iterative methods are parallelized is very challenging and choice of preconditioner is quite important. Referring back to Section 2.3.3, as at each optimization step iterative solution must be done and preconditioner should be calculated again (coefficient matrix changes as material distribution changes), calculation and application of chosen preconditioner heavily affect parallel performance of both iterative solution method and optimization problem. In earlier work, topology optimization of structural problems to optimize wings of airplanes under aerodynamic loads is studied and direct and iterative solvers are compared [21]. This work showed that direct methods such are usually not feasible to solve large scale problems with due to large need for memory resulting from fill-in. Also in [21], the CG paired with Block-Jacobi preconditioner in PETSc [5] is used for solution of large-scale problems, but parallel efficiency was low as increasing number of blocks (= increasing number of cores) causes a drop on quality of preconditioner.

Topology optimization for structural, heat transfer and potential flow problems is defined as;

$$
\begin{aligned}
\min_{\boldsymbol{\rho}} \quad & c(\boldsymbol{\rho}) = \boldsymbol{U}^T\boldsymbol{K}(\boldsymbol{\rho})\boldsymbol{U} \text{ in } V_o \\
\text{s. t.} \quad & V_f = \textstyle\sum_e \left(\rho_e V_e\right)/V_o \\
& \boldsymbol{K}(\boldsymbol{\rho})\boldsymbol{U} = \boldsymbol{F}(\boldsymbol{\rho}) \text{ in } V_o \\
& u = u_o \text{ on } \partial V_o \\
& 0 < \rho_e \leq 1
\end{aligned}
\tag{3.1}
$$

where , c is the objective function, V_e is the element volume, V_f is the target volume fraction of the design space, V_o is the initial volume of the design space, $\boldsymbol{\rho}$ is the design vector composed of element densities ρ_e (element volume fractions), $\boldsymbol{K}$ is the stiffness matrix of the system, $\boldsymbol{U}$ is the solution vector of field variables (e.g., displacements, temperature, velocity potential), $\boldsymbol{F}$ is the load vector, u_o is the boundary condition for u on the boundary ∂V_o. All variables are functions of the design variable $\boldsymbol{\rho}$, which is to be determined as the minimum point of c. The element properties is a function of the design variable defined as $E_e = \rho^p E_o$, where p is a penalty constant, E_o is the material property in the initial design space V_o, E_e is the modified property of the element e. For $p > 1$ intermediate volume fraction values are penalized. $p = 3$ is a common choise. The well-known optimality criteria method, which is a gradient-based method, [8] is used here. A special filtering technique for calculation of gradients [8] is used to prevent checker-board patterns. The third term in Eq. 3.1 is the linear system of equations which is to be solved repeatedly at each step of the optimization. Solving it efficiently and fast is important for the performance of the optimization method.

3.1.2 Multidisciplinary Examples

In following we present some practical applications of topology optimization for various disciplines. Even though physical applications, or physical meanings of solutions, may vary, mathematical foundation is the same. Initial design domain which is chosen to be a full for all cases, then they are gradually emptied at each optimization step to achieve target volume fraction. The regions found to be empty are defined to have small material density values rather than zero values (10^{-3} to avoid introducing a zero line or row to coefficient matrix).

Figure 3.1a is the result for when three-dimensional cube topology is optimized, under uniform load on the top and non slipping four bottom corners to reach the target volume fraction $V_f = 0.3$. Initial design space is the cube which is represented by black lines, brown color represents final design, empty regions are transparent. Given loading and boundary conditions final structure has minimum compliance energy and is a sound structure. Using denser meshes we can achieve higher resolution and capture more details which can be used to find microstructural designs. For this purpose, one should be able to solve large size problems efficiently.

Figure 3.1b is the result for the optimization of a heated domain where the right side is partially cooled and the rest insulated and the target volume ratio is $V_f = 0.2$. Near empty elements are shown by blue color, red and green colors denote, respectively, full and partially full elements. The topology optimization for thermal problems is to minimize the maximum temperature in the region by optimizing distribution of a given amount of an expensive conductive material. The tree-like structured structure with large amounts of details for this moderately dense mesh are also of interest.

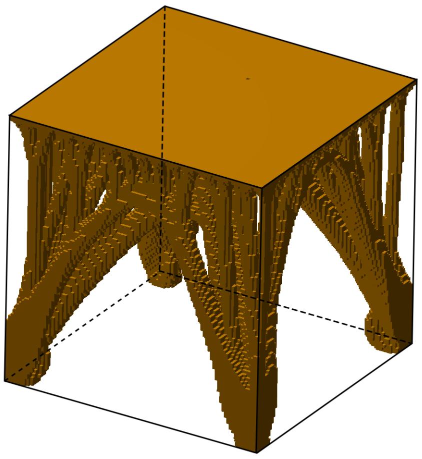
Figure 3.1c is the result for the optimization of a heated domain where its bottom center is the source of heat, the surface transfer heat convectively for cooling and the target volume ratio is $V_f = 0.3$. Near empty elements are shown by blue color, red and green colors denote, respectively, full and partially full elements. Here again, problem is to minimize the maximum temperature in the region by optimizing distribution of a given amount of an expensive conductive material.

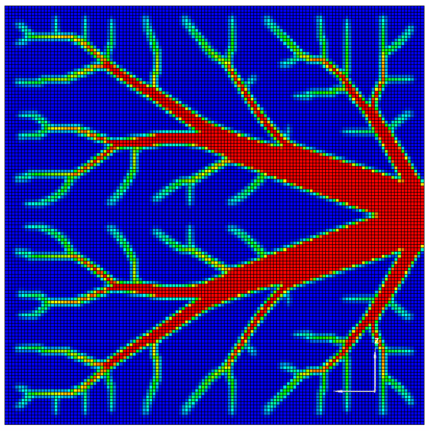
Figure 3.1d is the result for the optimization of a flow domain where the right side is inlet and there are two exits at the left side and the target volume ratio is $V_f = 0.3$. Again near empty elements are shown by blue color, red color denotes full elements, for this case the flow channel. Objective is find a flow network which limits pressure loss as much as possible and distributes velocities uniformly.

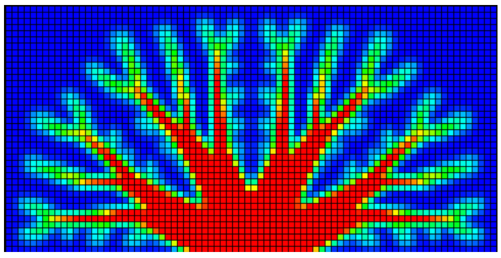
These problems require hundreds or thousands of optimization steps to reach the final topology. As problem size grows number of steps required increases. Results presented in Figure 3.1 are obtained and plotted using EDA's proprietary software CAEeda, http://www.caeeda.com.

3.2 ADVECTION-DIFFUSION-REACTION TYPE EQUATIONS

Advection-Diffusion-Reaction(ADR) type of equations are of great interest as they are used in modelling phenomena often. These kind of equations also sometimes called as convection–diffusion equation or drift–diffusion equation. Variable of interest may be concentration of species or heat. ADR type equations may be used to describe chemical phenoma or plasma simulations which may be used preparations before ac-


(a) A structural optimization problem


(b) A heat transfer optimization problem


(c) A convective heat transfer optimization problem

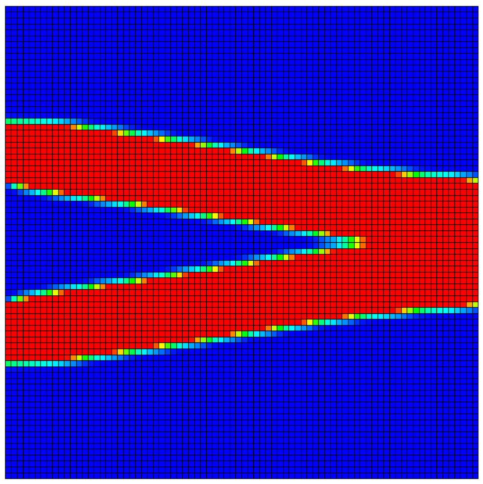

(d) A fluid flow-path optimization problem

Figure 3.1: Multidisciplinary topology optimization examples

tual experiments to reduce the costs of experiment by limiting cases of interests. These will accelerate design of new experiments and increase success rates. Both steady and unsteady (time-dependent) cases are important and quite challenging.

Three dimensional ADR equations share the curse of dimensionality. Linear systems are quite large and ill-conditioned, and direct solvers suffer from fill-in. If problem is unsteady (it is not considered in this thesis, but is interest of future work) successive solution of linear system of equations is necessary. This linear system may be changing or stay the same as time progresses depending on whether coefficients of ADR equation are time-dependent or not. Therefore, as in case with topology optimization problem, calculation and application of preconditioner heavily affects the performance.

3.2.1 3D formulation of the complete flux scheme

Let us consider a stationary conservation law of advection-diffusion-reaction type,

$$\nabla \cdot (\boldsymbol{u}\varphi - \varepsilon \nabla \varphi) = s, \tag{3.2}$$

where $\boldsymbol{u} = u\,\boldsymbol{e}_x + v\,\boldsymbol{e}_y + w\,\boldsymbol{e}_z$ is a mass flux or (drift) velocity, $\varepsilon \geq \varepsilon_{\min} > 0$ a diffusion coefficient, and s a source term describing, e.g., chemical reactions or ionization. The unknown φ is then the mass fraction of one of the constituent species in a chemically reacting flow or a plasma. The parameters ε and s are, for the sake of discretization, assumed to be functions of the spatial coordinates.

Let the flux vector $\boldsymbol{f}$ be, $\boldsymbol{f} := \boldsymbol{u}\varphi - \varepsilon\nabla\varphi$. Then equation 3.2 can be simplified as $\nabla \cdot \boldsymbol{f} = s$. Integrating this equation over a fixed domain Ω and applying Gauss's theorem the integral form of the conservation law is obtained,

$$\oint_\Gamma (\boldsymbol{f}, \boldsymbol{n})\, dS = \int_\Omega s\, dV, \tag{3.3}$$

where $\boldsymbol{n}$ is the outward unit normal on the boundary $\Gamma = \partial\Omega$. This equation is the basic conservation law, provided φ is smooth enough, is equivalent to 3.2. In the FVM [14] the domain is covered with a finite number of disjoint control volumes or cells and the integral form (3.3) is imposed on each one of these cells.

In three-dimensional Cartesian coordinates, first grid points $\boldsymbol{x}_{i,j,k} = (x_i, y_j, z_k)$ is chosen where solution for φ has to be found approximately. Next, control volumes $\Omega_{i,j,k} := (x_{i-1/2}, x_{i+1/2}) \times (y_{j-1/2}, y_{j+1/2}) \times (z_{k-1/2}, z_{k+1/2})$ are defined. Here $x_{i\pm1/2} := \frac{1}{2}(x_i + x_{i\pm1})$ etc. The boundary of control volume $\Omega_{i,j,k}$ is the union of six surfaces $\Gamma_{i\pm1/2,j,k}$, $\Gamma_{i,j\pm1/2,k}$ and $\Gamma_{i,j,k\pm1/2}$. Taking $\Omega = \Omega_{i,j,k}$ in conservation law (3.3) and approximating all integrals with the midpoint rule, we find

$$\begin{aligned} &\big(f_{x,i+1/2,j,k} - f_{x,i-1/2,j,k}\big)\,\Delta y\,\Delta z \\ &\qquad + \big(f_{y,i,j+1/2,k} - f_{y,i,j-1/2,k}\big)\,\Delta x\,\Delta z \\ &\qquad + \big(f_{z,i,j,k+1/2} - f_{z,i,j,k-1/2}\big)\,\Delta x\,\Delta y \doteq s_{i,j,k}\,\Delta x\,\Delta y\,\Delta z, \end{aligned} \tag{3.4}$$

where we have used that $\boldsymbol{f} = f_x\,\boldsymbol{e}_x + f_y\,\boldsymbol{e}_y + f_z\,\boldsymbol{e}_z$. Approximating the fluxes f_x, f_y and f_z we get F_x, F_y and F_z to find FVM (3.4).

Let us find the x-component of the flux, $f_{x,i+1/2,j,k}$, from solution of a local one-dimensional ordinary differential equation. Consider the flux $f := u\varphi - \varepsilon\frac{d\varphi}{dx}$ and the model BVP:

$$\frac{d}{dx}(f) = s, \quad x_i < x < x_{i+1}, \qquad \varphi(x_i) = \varphi_i, \quad \varphi(x_{i+1}) = \varphi_{i+1}. \tag{3.5}$$

$\varepsilon > 0$ and s are assumed to be sufficiently smooth functions of x. To solve (3.5) let the variables a, A and S be

$$a := \frac{u}{\varepsilon}, \quad A := \int_{x_{i+\frac{1}{2}}}^{x} a(\xi)\,d\xi, \quad S := \int_{x_{i+\frac{1}{2}}}^{x} s(\xi)\,d\xi, \tag{3.6}$$

and integrate (3.5) from the cell boundary $x_{i+\frac{1}{2}}$ to $x \in (x_i, x_{i+1})$ to find the integral relation $f - f_{i+\frac{1}{2}} = S$. By rewriting the flux in terms of its integrating factor, $f = -\varepsilon\,\mathrm{e}^{A}\frac{d}{dx}\left(\mathrm{e}^{-A}\,\varphi\right)$, and substituting it in the integral relation, then integrating over the interval (x_i, x_{i+1}), we get,

$$f_{i+\frac{1}{2}} = \underbrace{\frac{\mathrm{e}^{-A_i}\varphi_i - \mathrm{e}^{-A_{i+1}}\varphi_{i+1}}{\langle \varepsilon^{-1}, \mathrm{e}^{-A}\rangle}}_{f^{\mathrm{hom}}_{i+\frac{1}{2}}} \underbrace{- \frac{\langle \varepsilon^{-1} S, \mathrm{e}^{-A}\rangle}{\langle \varepsilon^{-1}, \mathrm{e}^{-A}\rangle}}_{f^{\mathrm{inh}}_{i+\frac{1}{2}}}. \tag{3.7}$$

To sum up, $f_{i+\frac{1}{2}} = f^{\mathrm{hom}}_{i+\frac{1}{2}} + f^{\mathrm{inh}}_{i+\frac{1}{2}}$. Here inner product is defined as $\langle f, g\rangle := \int_{x_i}^{x_{i+1}} fg\,dx$ when calculating the homogeneous flux f^{hom} and the inhomogeneous flux f^{inh}, which are the advection-diffusion operator and the source term, respectively. When u and ε are both constants,

$$A = \frac{u}{\varepsilon}\left(x - x_{i+\frac{1}{2}}\right), \quad \langle a, 1\rangle = \frac{u}{\varepsilon}\,\Delta x, \quad \langle \varepsilon^{-1}, \mathrm{e}^{-A}\rangle = \frac{1}{u}\langle a, \mathrm{e}^{-A}\rangle, \tag{3.8}$$

and f^{hom} reduces to the *constant coefficient flux*

$$f^{\mathrm{hom}}_{i+\frac{1}{2}} = \frac{\varepsilon}{\Delta x}\Big(B(-P)\varphi_i - B(P)\varphi_{i+1}\Big), \tag{3.9}$$

in which $B(x) := x/(\mathrm{e}^{x} - 1)$ and the Péclet number $P := \frac{u}{\varepsilon}\,\Delta x$. For the inhomogeneous flux we find

$$f^{\mathrm{inh}}_{i+\frac{1}{2}} = \int_0^1 G(\sigma; P)\,s(x(\sigma))\,d\sigma, \tag{3.10}$$

where we have used the *normalized coordinate* $\sigma(x) := (x - x_i)/\Delta x$ (note that $x(\sigma) := x_i + \sigma\,\Delta x$) and the *Green's function for the flux*

$$G(\sigma; P) := \begin{cases} \dfrac{\mathrm{e}^{-\sigma P} - 1}{\mathrm{e}^{-P} - 1}, & \text{for} \quad 0 \le \sigma \le 1/2, \\[2ex] -\dfrac{\mathrm{e}^{(1-\sigma)P} - 1}{\mathrm{e}^{P} - 1}, & \text{for} \quad 1/2 < \sigma \le 1. \end{cases} \tag{3.11}$$

For the numerical fluxes, two averages are used: the normal *arithmetic* average $\overline{\varepsilon}_{i+\frac{1}{2}} := (\varepsilon_i + \varepsilon_{i+1})/2$ and a *weighted* average $\widetilde{\varepsilon}_{i+\frac{1}{2}} := W(-\overline{P}_{i+\frac{1}{2}})\varepsilon_i + W(\overline{P}_{i+\frac{1}{2}})\varepsilon_{i+1}$. The weight function used here is $W(x) := (\mathrm{e}^x - 1 - x)/(x\,(\mathrm{e}^x - 1))$. A detailed derivation can be found in [25]. We use (3.9) to find the following *numerical homogeneous flux*

$$F^{\text{hom}}_{i+\frac{1}{2}} = \alpha_{i+\frac{1}{2}}\varphi_i - \beta_{i+\frac{1}{2}}\varphi_{i+1}, \tag{3.12a}$$

$$\alpha_{i+\frac{1}{2}} := B\big(-\overline{P}_{i+\frac{1}{2}}\big)\frac{\widetilde{P}_{i+\frac{1}{2}}}{\overline{P}_{i+\frac{1}{2}}}\frac{\widetilde{\varepsilon}_{i+\frac{1}{2}}}{\Delta x}, \quad \beta_{i+\frac{1}{2}} := B\big(\overline{P}_{i+\frac{1}{2}}\big)\frac{\widetilde{P}_{i+\frac{1}{2}}}{\overline{P}_{i+\frac{1}{2}}}\frac{\widetilde{\varepsilon}_{i+\frac{1}{2}}}{\Delta x}. \tag{3.12b}$$

The *numerical inhomogeneous flux* is based on (3.10). We take $s(x)$ equal to s_i on the interval $(0, 1/2)$ and equal to s_{i+1} on $(1/2, 1)$. Next we integrate the Green's function to find

$$F^{\text{inh}}_{i+\frac{1}{2}} := \gamma_{i+\frac{1}{2}}s_i - \delta_{i+\frac{1}{2}}s_{i+1}, \tag{3.13a}$$

$$\gamma_{i+\frac{1}{2}} := \Delta x \int_0^{1/2} G(\sigma; \overline{P}_{i+\frac{1}{2}})\,d\sigma = C(-\overline{P}_{i+\frac{1}{2}})\,\Delta x, \tag{3.13b}$$

$$\delta_{i+\frac{1}{2}} := \Delta x \int_{1/2}^{1} G(\sigma; \overline{P}_{i+\frac{1}{2}})\,d\sigma = C(\overline{P}_{i+\frac{1}{2}})\,\Delta x, \tag{3.13c}$$

in which $C(x) := (\mathrm{e}^{\frac{1}{2}x} - 1 - \frac{1}{2}x)/(x\,(\mathrm{e}^x - 1))$. Note: $C(x) \to 1/8$ for $x \to 0$, $C(x) \to 0$ for $x \to \infty$, and $C(x) \to 1/2$ for $x \to -\infty$. Notice that γ and δ are almost equal for small Péclet numbers and the inhomogeneous flux is small. The upwind value of s is dominantly effective for large (positive or negative) Péclet numbers. Adding (3.12) and (3.13), *numerical complete flux* is found:

$$F_{i+\frac{1}{2}} = \alpha_{i+\frac{1}{2}}\varphi_i - \beta_{i+\frac{1}{2}}\varphi_{i+1} + \gamma_{i+\frac{1}{2}}s_i - \delta_{i+\frac{1}{2}}s_{i+1}. \tag{3.14}$$

By combining the one-dimensional schemes, we will extend FVM scheme to the 3D equation (3.2). Including the cross-fluxes $\partial f_y/\partial y$ and $\partial f_z/\partial z$ in the computation of the flux in x-direction is the main idea . By doing this, we can reduce the crosswind diffusion and find much sharper layers for advection-dominated flows. In [25] we have shown that for 2D problems, the inclusion of cross-flux terms is essential to maintain second order accuracy, whereas the homogeneous flux scheme (without cross-fluxes) reduces to first order. We can find the numerical flux $F_{x,i+\frac{1}{2},j,k}$ from the quasi-one-dimensional boundary value problem:

$$\frac{\partial}{\partial x}\Big(\Big(u\varphi - \varepsilon\frac{\partial \varphi}{\partial x}\Big)\Big) = s_x, \qquad x_i < x < x_{i+1},\ y = y_j,\ z = z_k, \tag{3.15a}$$

$$\varphi(\boldsymbol{x}_{i,j,k}) = \varphi_{i,j,k}, \qquad \varphi(\boldsymbol{x}_{i+1,j,k}) = \varphi_{i+1,j,k}, \tag{3.15b}$$

where the modified source term s_x is defined by $s_x := \alpha s - \beta(\partial f_y/\partial y + \partial f_z/\partial z)$, with α and β coefficients that are yet to be determined. If we take $\beta = 1$, the cross-fluxes are completely included; taking $\beta = 0$ ignores them.

The numerical flux is similar to derive as in the case of (3.14), the main difference is the inclusion of the cross-fluxes $\partial f_y/\partial y$ and $\partial f_z/\partial z$ in the source term. To compute

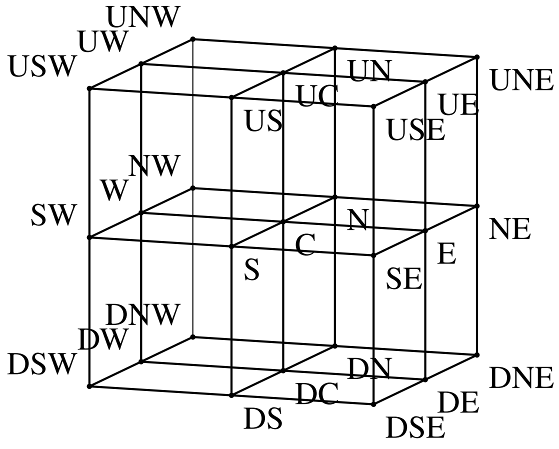


Figure 3.2: Compass notation for points in discretization stencil.

s_x, $\partial f_y/\partial y$ is replaced by its central difference approximation and for f_y we take the homogeneous numerical flux. We treat $\partial f_z/\partial z$ in the same way. Similar procedures apply to the y- and z-components of the flux. We shall take $\beta = 1/2$ in the numerical simulations. Adding the three one-dimensional problems in x, y and z-direction, we find that we need to choose $\alpha = (1 + 2\beta)/3$ for consistency. The numerical fluxes presented above are substituted into (3.4) and we find 27-point stencil for the unknown φ. The points of the stencil are presented in Figure 3.2. We denote the resulting linear system by $\boldsymbol{Ax} = \boldsymbol{b}$. The matrix $\boldsymbol{A}$ has in general 27 nonzero diagonals.

The resulting sparse linear system from the 3D discretization is most suitable for iterative solvers since direct solvers are known to scale poorly and memory requirements are usually very high due to fill-in. We use Bi-Conjugate Gradient Stabilized (BiCGStab) with preconditioning. We will study and compare the performance of two parallel preconditioners that are based on the sparse approximate inverse (SAI) and incomplete LU (ILU) factorization.

3.2.2 A three-dimensional flow problem

We consider the following flow problem as our test problem. It is a three-dimensional extension of the problem in Section 8, Example 3 of [25]. The problem domain is given by $-1 \leq x \leq 1$, $0 \leq y \leq 1$ and $0 \leq z \leq 1$.

$$\nabla \cdot (\mathbf{u}\varphi - \varepsilon \nabla \varphi) = 0, \qquad \text{in } (-1,1) \times (0,1) \times (0,1) \tag{3.16}$$

with velocity field $\mathbf{u}(x,y,z) = (1-x^2)y(1-2z)\,\mathbf{e}_x + x(1-y^2)(1-2z)\,\mathbf{e}_y + 4xyz(1-z)\,\mathbf{e}_z$, see Figure 3.3. We impose the following boundary conditions (c is a constant that determines the steepness of the profile; we take $c = 10$)

- At the inlet (given by $y = 0$, $x \geq 0$, $0 \leq z \leq 1/2$ and $y = 0$, $x \leq 0$, $1/2 \leq z \leq 1$) we set $\varphi(x,y,z) = \big(1 + \tanh(c(2x+1))\big)z$

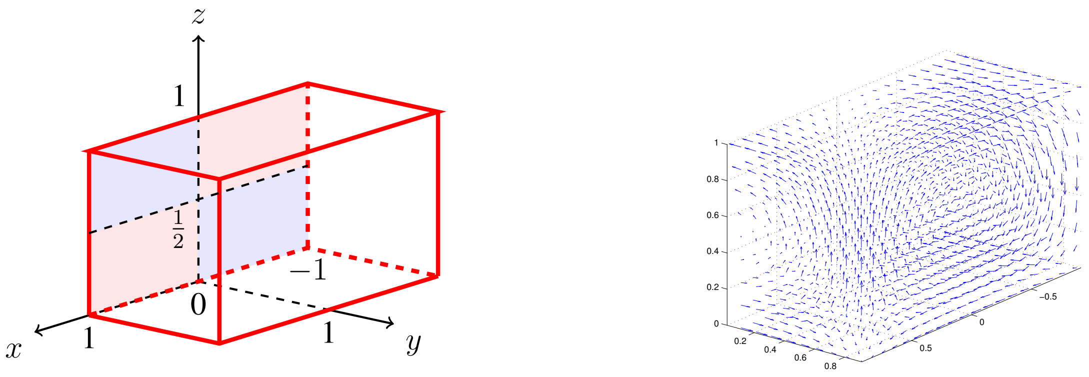


Figure 3.3: The problem domain and velocity field for the flow problem. The inlet is shaded red; the outlet blue

- At the outlet (given by $y = 0$, $x < 0$, $0 \leq z < 1/2$ and $y = 0$, $x > 0$, $1/2 < z \leq 1$) we set $\frac{\partial \varphi}{\partial y}(x, y, z) = 0$
- At the front ($x = 1$), back ($x = -1$), right ($y = 1$), bottom ($z = 0$), and top ($z = 1$), we set $\varphi(x, y, z) = \big(1 + \tanh(c(2x + 1))\big)(1 - y)z$.

CHAPTER 4

RESULTS

In this chapter, we present the performance and efficiency of proposed methods of solution against some other widely used methods. The problems we consider are, even though from different formulations, similar in sense of sparsity, bandedness and being ill-conditioned. In both cases, linear systems to be solved may become very large depending on required accuracy. Thus, solving these using direct solution methods becomes increasingly infeasible. Iterative methods may not converge as methods to solve problems usually result in ill-conditioned linear systems. Preconditioners are required for iterative methods to converge. But computation and application of preconditioners are extra costs which are not included in direct solution methods. Therefore, for an iterative method to be feasible over direct method, an easy to compute, easy to apply and at the same time good preconditioner is necessary. On the two problems we consider, we show that our proposal is a good candidate.

We want to emphasize that, beside those problems mentioned above, it is important for a method to be parallelizable. Even though a method may prove itself to be faster than any other algorithm in sequential sense, it may take too long time to be considered feasible. A method that has long computational time(wall clock time) in sequential, but scales well in parallel, may be made run shorter by using more computational units (nodes, cores). This is very crucial in both problems that we consider.

Results and figures in this chapter are (accepted) to be published in [1, 24].

4.1 Computing Environments

Throughout numerical experiments presented in this thesis, platforms presented in Table 4.1 are used for computations. While Avokado and Greyfurt are one node, shared memory parallel machines, NAR is a (multiple nodes, distributed memory) cluster. All computers are located at the Department of Computer Engineering, Middle East Technical University and NAR is provided by same department again.

Table 4.1: Parallel computing platforms

	OS	Processors	RAM
Avokado	CentOS 6.6	2 x Intel Xeon E5-2650v3	64 GB
Greyfurt	Debian Wheezy	AMD Opteron 6376	128 GB
NAR	Scientific Linux v5.2 64-bit	46 x 2 x Intel Xeon E5430	16 GB per node

4.2 Performance for Topology Optimization

It is noted at the start of this section, problems we consider are similar in nature. On the contrary to that, a difference between those is that Topology Optimization problems requires solution of linear systems successively. This poses a great challenge for any solution method that may be suggested. For any method to solve Topology Optimization problems, it is most important to be as much as parallelizable to shorten total time spent.

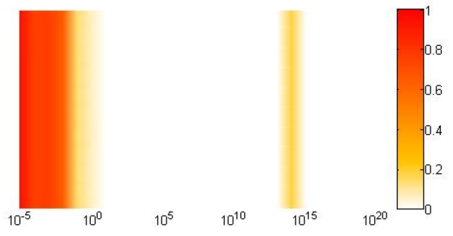
Usually these matrices are very ill-conditioned. Before starting to solve the linear system itself, scaling is required to enhance eigenvalue distribution. Let us consider a heat topology optimization problem with 100.000 elements. Solution to the problem can be seen in Figure 3.1c. Note that, coefficient matrix for this problem is a symmetric matrix of size 112.221. Eigenvalue distribution of original problem (Figure 4.1a) is undesirable due to accumulation of eigenvalues around 0 and eigenvalues of order 10^{14}. Scaling as seen in Figure 4.1b make eigenvalue distribution by eliminating eigenvalues of large order. Symmetric diagonal scaling ($\widetilde{\boldsymbol{A}} = \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$) is applied to preserve symmetry of original matrix. As symmetry is preserved we still can use CG as iterative solver rather than BiCGStab which introduces a computational overhead. Lastly applying SPAI preconditioner Figure 4.1c gathers eigenvalues around 1. This way we can solve, otherwise unsolveable, problems by iterative solvers.

As small scale experiment, we have compared three methods in shared memory environments (Avokado and Greyfurt) for same heat transfer problem we have presented eigenvalue distributions for. These three methods are ParaSails preconditioner paired with CG, Block-Jacobi preconditioner with CG and Pardiso, direct solver. Stopping criteria for CG iterations is either the relative residual norm drops below 10^{-8} or maximum number of iterations 10000 is reached. In Table 4.2 sequential solve times of different methods are given.

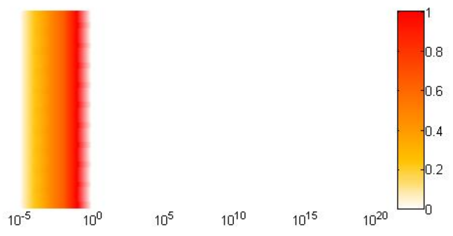
Table 4.2: Total time spent for sequential solve in seconds

	ParaSails	Block Jacobi	Pardiso
Avokado	2,37	1,19	4,38
Greyfurt	6,56	2,75	10,18

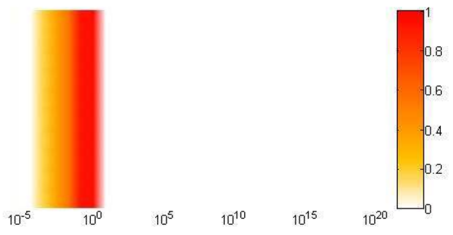
In Figure 4.2, we present speedups of different methods with respect to increasing



(a)

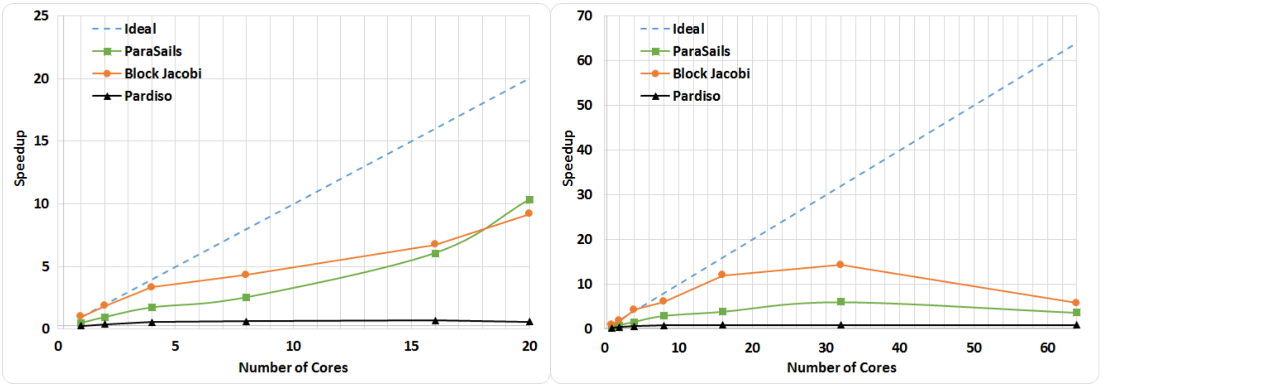


(b)

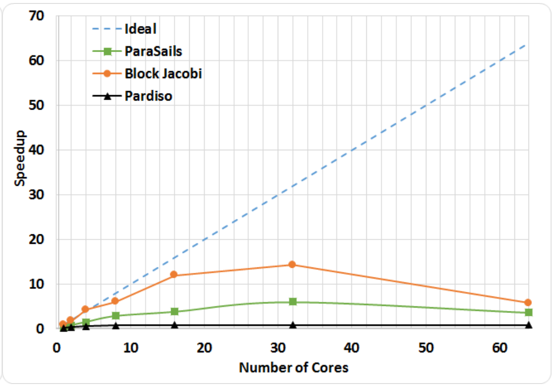


(c)

Figure 4.1: Eigenvalue distributions



(a) Speedup obtained in Avokado



(b) Speedup obtained in Greyfurt

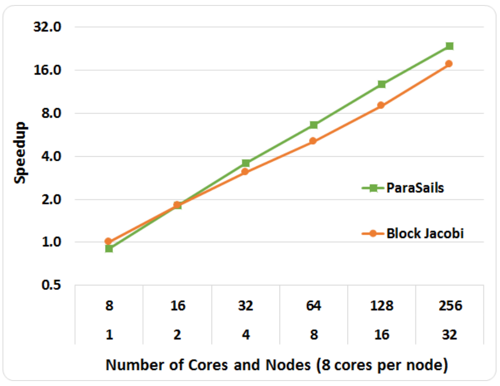
Figure 4.2: Speedup graphs for small-scale problem

number of processors. Speedups are calculated by;

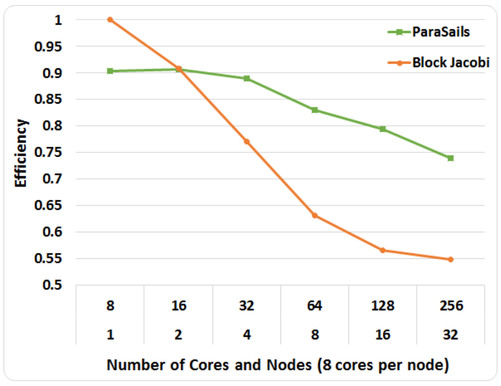
$$\text{Speedup} = \frac{\text{Best Sequential Time}}{\text{Time with n number of cores}}. \tag{4.1}$$

Ideal speedup would be equal to number of workers. But it usually is not possible due to memory bandwidth limitations or communication between processes. Speedups are calculated with respect to Block-Jacobi sequential time (application of preconditioner is done using iterative solver without preconditioning) as on both platforms it is the best time. On Avokado, when 20 cores are used, ParaSails is faster than Block-Jacobi. On Greyfurt both has poor scalability, which is possibly resulting from memory bandwidth limitations of platform, and throughout Block-Jacobi is better.

For large scale experiment, scalability of methods across multiple nodes of NAR cluster is studied and compared. This linear system is of size 3.090.903 which with



(a) Speedup graph for large-scale problem



(b) Efficiency graph for large-scale problem

Figure 4.3: Node based speedup and efficiency graphs for large-scale problem

248.529.012 nonzeros. This linear system results from the structural problem with 1.000.000 elements, solution is visualized in Figure 3.1a. Similar to small scale experiment, we have compared CG with ParaSails preconditioner and CG with Block-Jacobi Preconditioner. Stopping criteria for CG is to stop when relative residual norm is below 10^{-8}. Figure 4.3a presents the speedup of methods with respect to best solution time achieved on a single node, in this case CG paired with Block-Jacobi preconditioner and Figure 4.3b shows efficiency which is Speedup divided by Number of cores. We have chosen this method of representation as inside a node speedup is low because of memory bandwidth limitations. If all cores of a single node are used, due to limitations speedup is around 2,3. Load on the memory bandwidth can be relaxed, for example by using less cores from each node and distributing load to cluster, and in turn better speedup is achieved (around 3,6) for same number of cores (8 cores, 2 cores from 4 nodes). This is not a reasonable choice as we do not use valuable resources for the sake of speedup. When all cores in nodes used, Block Jacobi preconditioner seems to be best choice for one node, as number of nodes increases its performance worsens and ParaSails preconditioner scales almost linearly for large number of nodes.

Note that reference solve time is for one node. CG with ParaSails spends 3.038 seconds and CG with Block-Jacobi requires 2.746 seconds.

The number of block-Jacobi iterations appears to remain constant as number of cores increases from 128 to 256. This is expected as block size gets smaller Block-Jacobi preconditioners start converge diagonal preconditioner. Difference between them gets smaller. As we are using a scaled version of the linear system, the diagonal preconditioner is actually identity matrix. This is equivalent to solving linear system without a preconditioner, in which case CG diverges. Therefore, in theory, as number of blocks increases (equivalently, block size gets smaller) number of iterations required for convergence increases, even though it may be slow.

For all test problems, we have found out CG method when paired with ParaSails precondtioner gives good performance. Block-Jacobi preconditioner is used for comparison as it was used in an earlier work [21]. Pardiso is used for comparison against direct solution methods. On distributed memory platforms scalability of CG with ParaSails

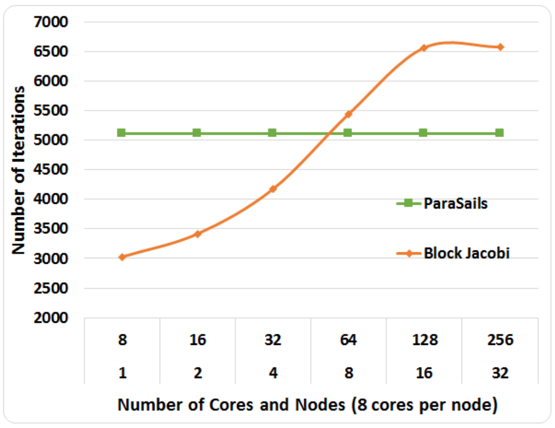


Figure 4.4: Number of iterations vs. Number of cores

preconditioner is almost linear.

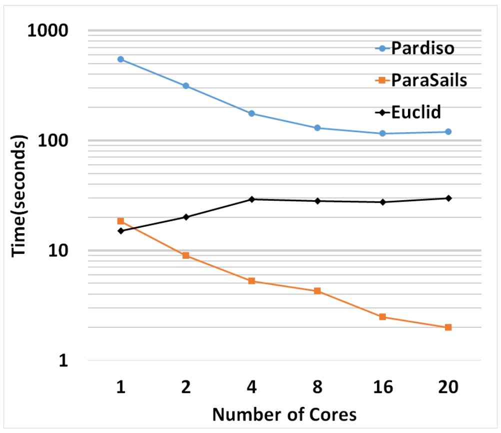
4.3 Performance for 3D ADR-type Problems using Complete-Flux scheme

All runs for this problem are done on Avokado. Hypre ([15]) version `2.10.0b` is used for solution. It provides parallel environment for variety of iterative solvers and preconditioners using MPI. We choose preconditioned BiCGStab [26], since when problem is advection dominated linear system is not symmetric, as the Krylov subspace method. We tested two preconditioners, namely, Euclid [18] and ParaSails [12]. Euclid is a parallel ILU implementation and ParaSails is a parallel SAI type preconditioner. Euclid supports Parallel-ILU(k) and Block-Jacobi ILU. We use Parallel-ILU(1). For ParaSails, parameters are set so that in the worst case the preconditioner has same nonzero structure as coefficient matrix. Nonzeros of coefficient matrix, which are less than 0,2 in absolute value, are ignored when determining sparsity structure of preconditioner. Also after computing preconditioner, nonzeros of preconditioner which are less than 0,05 in absolute value are dropped.

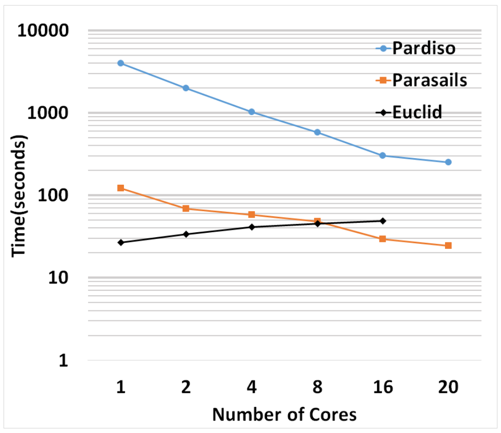
Pardiso, with default parameters, is used for comparison against direct solver.

Two different systems with same size are used for the following experiments. They result from different choices for diffusion coefficients of 1 and 10^{-5}. Initially systems are generated including boundary conditions. To reduce the size of system, Dirichlet boundary conditions are removed. After this step, system are of size 992.319 unknowns and, have 18.467.751 and 17.452.253 nonzeros respectively for diffusion coefficients of 1 and 10^{-5}. Smaller diffusion coefficient results in sparser coefficient matrix. For BiCGStab, the stopping criterion is set to be either relative residual norm drops below 10^{-8} or maximum number of iterations (10.000) is reached. Each MPI process is mapped on a single physical core.

Table 4.3 shows change of number of iterations for different diffusion coefficient with respect to number of cores. When ParaSails is used, the number of iterations are independent of the number of cores, even though large. Euclid requires more iterations



(a) $\varepsilon = 1$



(b) $\varepsilon = 10^{-5}$

Figure 4.5: Parallel running time in seconds

as the number of cores increases, and for diffusion coefficient of 10^{-5} BiCGStab fails when used with Euclid.

Table 4.3: Number of iterations for the iterative solvers using different diffusion (ε) coefficients

	$\varepsilon = 1$		$\varepsilon = 10^{-5}$	
Cores	ParaSails	Euclid	ParaSails	Euclid
1	226	44	1178	94
2	212	44	1183	125
4	215	49	1186	164
8	236	45	1191	235
16	220	51	1151	415
20	213	52	1177	-

Figure 4.5 shows solution times for Pardiso, BiCGStab with ParaSails and BiCGStab with Euclid as number of cores increases. As expected Pardiso is the slowest due to fill-in during the factorization stage. Euclid is faster for 1 core but for larger number of cores it gets slower. This is probably due increasing number of iterations and inherently sequential time of triangular solves. ParaSails both have best time for large number of cores and best scalability.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In science and engineering (that is, theoretical and practical applications), partial differential equations are important tools for modelling. As dimension of problem increases (even though more than three dimensions in space may be meaningless in physical world) numerical solution of the PDEs become infeasible (because of time spent solving) and impossible (because of usually high condition number of resulting linear system). We have investigated direct and iterative solution methods for solution of such systems. With the introduction of multi and many core architectures, parallelism is one way to shorten the simulation time. We have studied how one can improve the parallel scalability and the number of iterations using various preconditioning techniques.

We have confirmed the poor performance of direct solvers due to fill-in. Iterative methods while having better performance, may fail. This problem is not unique to iterative solvers, but for direct solution methods there are precautions like pivoting, iterative refinement of solution by default even though user may not be aware of. Preconditioners are remedy for this problem in case of iterative methods. We have applied different types of preconditioners to two important problems. Results show that, iterative methods when paired with SAI type preconditioners are both scalable and have relatively much better performance.

Future work consists diverse interests. Firstly, during experiments we noticed that due to memory bandwidth limitations of systems performance suffers in shared memory architectures. This can possibly be prevented using threads rather than message passing inside each node and message passing between nodes. For example, OpenMP and MPI together in a hybrid model can be used and also MPI-3 has routines to achieve this. In both cases if PETSc or some other library is being used to accelerate implementation, there may be modifications those has to be done on libraries. Recomputation of preconditioner from scratch at each iteration in case of topology optimization problems is also great computational overhead, it may be possible to reuse same preconditioner with small changes if coefficient matrix does not change significantly (which is observed as optimization iteration progresses). Extension of three dimensional complete flux scheme to time-dependent problems and solution is another interest.

REFERENCES

[1] H. U. Akay, E. Oktay, M. Manguoglu, and A. A. Sivas, Improved parallel preconditioners fo multidisciplinary topology optimizations, International Journal of Computational Fluid Dynamics.

[2] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, MUMPS: MUltifrontal Massively Parallel Solver, version 2.0, Report TR/PA/98/02, 1998.

[3] W. E. Arnoldi, The principle of minimized iterations in the solution of the matrix eigenvalue problem, Quarterly of applied mathematics, 9(1), pp. 17–29, 1951.

[4] O. Axelsson, A survey of preconditioned iterative methods for linear systems of algebraic equations, BIT Numerical Mathematics, 25(1), pp. 165–187, 1985.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, PETSc Web page, `http://www.mcs.anl.gov/petsc`, 2015.

[6] R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43, Siam, 1994.

[7] M. Bendsoe and N. Kikuchi, Generating optimal topologies in structural design using a homogenization method, Computer Methods in Applied Mechanics and Engineering, 71(2), pp. 197 – 224, 1988, ISSN 0045-7825.

[8] M. P. Bendsøe and O. Sigmund, *Topology Optimization: Theory, Methods and Applications*, Berlin, Springer, second edition, 2003.

[9] M. Benzi, Preconditioning techniques for large linear systems: a survey, Journal of computational Physics, 182(2), pp. 418–477, 2002.

[10] T. Borrvall and J. Petersson, Topology optimization of fluids in stokes flow, International Journal for Numerical Methods in Fluids, 41(1), pp. 77–107, 2003, ISSN 1097-0363.

[11] T. E. Bruns, Topology optimization of convection-dominated, steady-state heat transfer problems, International Journal of Heat and Mass Transfer, 50(15-16), pp. 2859–2873, July 2007.

[12] E. Chow, Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns, International Journal of High Performance Computing Applications, 15(1), pp. 56–74, 2001.

[13] J. Deaton and R. Grandhi, A survey of structural and multidisciplinary continuum topology optimization: post 2000, Structural and Multidisciplinary Optimization, 49(1), pp. 1–38, 2014, ISSN 1615-147X.

[14] R. Eymard, T. Gallouët, and R. Herbin, Finite volume methods, in P. G. Ciarlet and J. L. Lions, editors, *Handbook of numerical analysis*, volume VII, pp. 713–1020, Elsevier, North-Holland, 2000.

[15] R. Falgout and U. Yang, hypre: A library of high performance preconditioners, Computational Science—ICCS 2002, pp. 632–641, 2002.

[16] M. J. Grote and T. Huckle, Parallel preconditioning with sparse approximate inverses, SIAM Journal on Scientific Computing, 18(3), pp. 838–853, 1997.

[17] A. Gupta, Wsmp: Watson sparse matrix package (part-i: direct solution of symmetric sparse systems), IBM TJ Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC, 21886, 2000.

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

[19] X. Li and J. W. Demmel, Superlu dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Trans. Mathematical Software, 29, pp. 110–140, 2003.

[20] M. Manguoglu, *Parallel Solution of Sparse Linear Systems*, pp. 171–184, Springer London, London, 2012, ISBN 978-1-4471-2437-5.

[21] E. Oktay, H. Akay, and O. Sehitoglu, Three-dimensional structural topology optimization of aerial vehicles under aerodynamic loads, Computers & Fluids, 92, pp. 225 – 232, 2014, ISSN 0045-7930.

[22] Y. Saad, *Iterative methods for sparse linear systems*, Siam, 2003.

[23] O. Schenk and K. Gärtner, Solving unsymmetric sparse systems of linear equations with pardiso, Future Gener. Comput. Syst., 20(3), pp. 475–487, April 2004, ISSN 0167-739X.

[24] A. A. Sivas, M. Manguoglu, J. H. M. ten Thije Boonkkamp, and M. J. H. Anthonissen, Discretization and iterative schemes for advection-diffusion-reaction problems, Lecture Notes in Computational Science and Engineering, Numerical Mathematics and Advanced Applications - ENUMATH 2015.

[25] J. H. M. ten Thije Boonkkamp and M. J. H. Anthonissen, The finite volume-complete flux scheme for advection-diffusion-reaction equations, Journal of Scientific Computing, 46(1), pp. 47–70, 2011.

[26] H. A. van der Vorst, Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems, SIAM Journal on Scientific and Statistical Computing, 13(2), pp. 631–644, 1992.