Solver Schemes for Linear Systems
Oral Comprehensive Exam Position Paper

Kanika Sood
Department of Computer and Information Science
University of Oregon

December 9, 2016

1 Introduction

This report presents different approaches to solving sparse linear systems—direct and iterative solvers—and compares a number of methods in each category. I also consider the two techniques of using solvers for linear systems: single method solver schemes and multi-method solver schemes. I also present some of the popular systems that use these techniques. This involves surveying the relevant literature that contains in-depth information about techniques in the field.

Linear systems are a form of representation for problems in a variety of domains, including but not limited to statistics, thermodynamics, electric circuits, quantum mechanics, nuclear engineering, fossil fuels, robotics and computational fluid dynamics. Solution methods in numerical optimization and PDE-based simulations frequently rely on the efficient solution of large, typically sparse linear systems. Given that linear systems are widespread [75, 74] across different areas of research, providing accurate and efficient solution methods plays a critical role for scientists in these fields. There are two classes of linear systems: sparse and dense. The systems in which most of the elements are zero are known as sparse systems. Those systems for which most of the elements are non-zero are referred to as dense linear systems. A common approach of storing sparse matrices requires only the non-zero elements to be stored, along with their location, instead of storing all the elements including the zero elements. Large sparse linear systems tend to represent real world problems better than dense systems and large sparse systems occur more frequently than small or dense systems. In this report, I focus on sparse linear systems.

Large sparse linear systems arise in many computational problems in science and engineering. Over the last several decades, applied mathematicians and computer scientists have developed multiple approaches to solving such linear systems. The traditional approach involves using a single solver, possibly combined with a preconditioner to get the solution. Preconditioning is discussed in more detail in Section 4. This solver can be chosen among a number of available options. For sparse linear systems, iterative solvers would be a more reasonable choice than direct solvers [30] because direct solvers can be more computationally expensive and might not even produce a solution. The challenge then is to identify which solver to use among the numerous options available because it is nontrivial even for experts to determine which is the best solver. The main reason is that the best solution is not consistent for a variety of problems occurring in different domains, or even different problems from the same domain. In addition, given that these are iterative solutions, which are

approximations of the solution, there may be more than one acceptable solution, which further adds to the complexity of choosing a solver. Another challenge in single-solver solutions is the reliability of getting a solution from one single solver. Consider the situation where the chosen solver fails to provide a solution, which defeats the purpose of a model that plans to offer solutions for its users. These problems motivate a different methodology of solving the given problem, namely, an approach that does not depend on a single method to get the solution for the system.

The second approach involves using multiple solvers (a composite of suitable solvers) [11, 73, 69, 32], instead of a single solver. If two or more solvers are used instead of one, the chances of getting to a solution increase. Further, there are different techniques of using multiple solvers for solving sparse linear systems: composite solvers, adaptive solvers and poly-iterative solvers. These are discussed in detail later in this section. The first technique has the solvers arranged in a sequence. It picks the first solver in the order and tries to solve the system; if this solver fails, it uses the next solver in the list. The second technique uses only one solver, but which solver would be used is decided dynamically. The third technique solves the system with multiple solvers simultaneously. As soon as a solver gets to a solution, the computation by the rest of the solvers is terminated.

The major advantage of using multiple solvers over the traditional single-solver technique is improving the reliability and providing significant performance improvement.

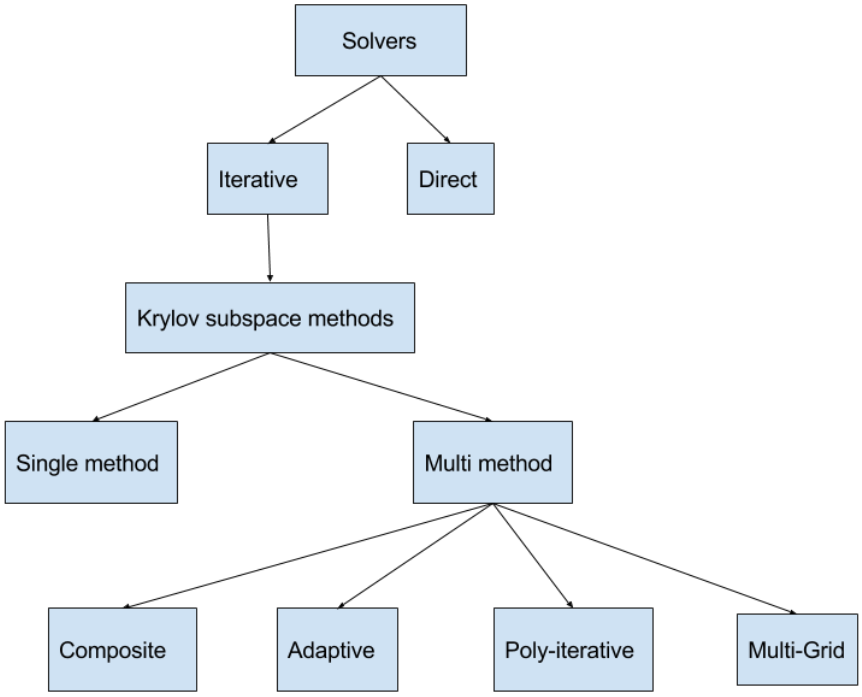

Figure 1: Solver hierarchy

Many modeling and simulation problems require solutions of large sparse linear systems, hence scientists from different domains, with different levels of expertise in the domain and programming background must be able to use linear solvers effectively. To select a solution method, the application developer must read all necessary documentation, be an expert programmer and should have done an analysis on the kinds of solver methods available for his problem. Practically considering the time required to read various documents and expecting all these skills from researchers with backgrounds in non-computer science fields, is not feasible. Even if they decide on a solver method, the chances that the method will be able to optimally solve the system, or in fact even be able to solve the system, are low. Selecting a suitable solver is not the only concern; in addition, the reliability needs to be improved. Problems from different domains may have different characteristics and requirements; in fact, a single domain may have problems with a variety of characteristics, which makes it harder to make guesses for what an optimal solver method would be, or to set a single solver method as default.

There has been some work done in this area in the past; the scope and consequence of using multiple solvers instead of a single solver was identified in 1968 [62], but still there is work to be done. Performance and reliability can be potentially improved by using a combination of solution methods instead of a single-solver scheme. In the case of a single-solver scheme, the complexity of choosing a suitable solver arises from two factors. First, the expertise required to make the decision of the solution method is not common in researchers of different fields. Second, the chance that solver method will continue to remain suitable with change of problems is minimal. This is because of the fact that as the problem changes, the problem characteristics also change; hence the good solving method(s) also change for the problem. With a variety of solution methods available for any given problem, there can be more than one good method to solve the system. Instead of choosing from those methods, if a scheme considers using multiple methods, the likelihood of successfully solving the system increases. Such a scheme will be referred to as multi-method scheme in the future scope of this document. In this document I review prior research in which either of these two solving schemes has been applied. Figure 1 shows the hierarchy of the sparse linear solvers, each of which is discussed in detail in the later sections.

This document surveys the categories of solvers and the two popular schemes applied for solving linear systems in detail and various kinds of systems that follow these scheme. This document is structured as follows. The next section describes the motivation behind this work. The following section outlines the single-method solver approach and presents the categories of solvers available for solving linear systems, namely direct and iterative, focusing on the Krylov subspace methods. In addition, I present a comparison of direct and iterative solvers. Section 4 discusses the pre-conditioning process. Section 5 outlines the parallelism issues for the solver methods from both solver categories and introduces multigrid methods. In the following section, multi-method solver approaches are discussed in detail. Section 7 presents some of the software packages available for various solvers. Section 8, elaborates on the common approaches for performance modeling for solvers. Section 9 delivers the conclusions and outlines the questions we would like to address and other tasks that can be done in the future. In the last section, I provide the appendix for this report.

2 Motivation

Systems of linear equations arise in many scientific and engineering computations. The use of linear systems is not just limited to scientists, in fact they are present in our day-to-day lives

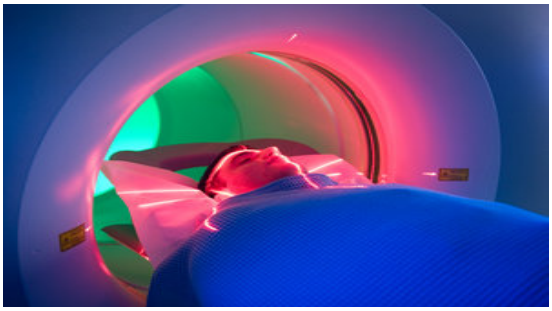

Figure 2: Head Scan chamber [a]

[a]Source: HD imagelib online resources

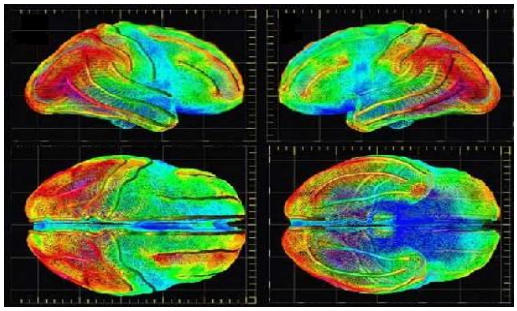

Figure 3: Head scan of a patient [a]

[a]Source: Austin CC online class resources

as well. This section presents a few examples to demonstrate how linear systems are widespread in various domains and how their solutions impact our daily lives. Solutions of linear systems therefore, are crucial for a much wider range of audiences. They are used in electric networks, balance network reactions, weather modeling, etc. Consider an imaging chamber for a head scan. Figure 2 shows the air pressure in different areas of the chamber, which emit and consume waves of different wavelengths. Different colors here are the different wavelengths of the rays, which construct images of the human brain from different angles to detect mild physiological changes in the brain and also anomalies, including bleeding, tumor, etc. Figure 3 shows the CAT scan of a patient, which is a collection of slices of human brain, where each slice can be represented as a set of linear equations. Each slice is a reconstructed image captured by the scanner. The pictures offer an image of the density of tissue in different parts of the brain.

Traffic flow in a network of streets is yet another common example of linear systems in day-to-day life. Decisions such as when the road surface will wear down or how the traffic lights should be set up depends on the flow rate of cars in each segment of streets in the given area. Figure 4 shows the university area for University of Oregon. The streets with the arrows are the one-way streets showing the direction of vehicles. Consider the area in the red box. This part on the map shows the area we are interested to monitor for flow rate of cars. This can be in the form of linear equations, where the number of cars entering and exiting different streets and the ones entering this whole red area can form a set of linear equations.

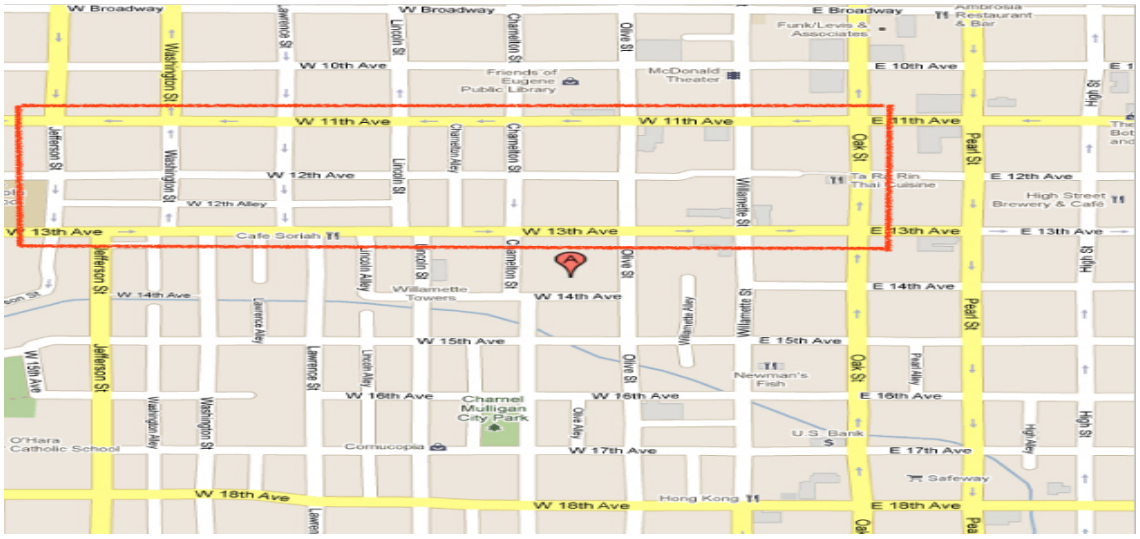

Figure 4: Traffic flow in a network of streets near University of Oregon.

Another example of linear systems is a room thermostat system behavior. The control system is modeled as differential equations and solution of these equations helps the system designer to design the thermostat in such a way that it responds fast to temperature change. The solution to these equations helps make the decision for smoother functionality of the thermostats. As a result, the transition from ON state to OFF state and vice-versa is more accurate. The two states of the thermostat are shown in Figures 5 and 6.

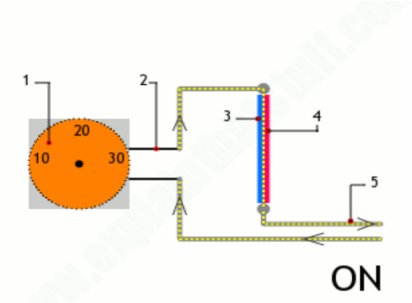

Figure 5: Thermostat in ON stage, with the circuit being complete.
[a]

[a]Source: Web

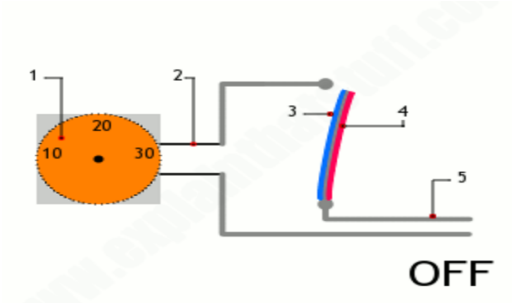

Figure 6: Thermostat in the OFF stage, with the circuit being broken.
[a]

[a]Source: Web

One other instance of linear equations, as illustrated in [1] is in structural engineering for deflection of membranes and beam-bending problems. These are differential equations, which are discretized by finite elements to obtain linear systems which are then solved to obtain solutions for

these problems. Therefore linear systems not only solve large problems, but are also useful in small problems arising from various scenarios like those mentioned above.

As programming techniques and computer capabilities improve, the size of systems that can be solved is also increasing. With the advancement in understanding of solver methods, small-sized systems can be solved very easily; in fact most (or all) of the solver methods that can give an exact solution of the system are able to solve these problems, with varying solve time. Solving large sparse systems brings its own challenges though. The systems, being large usually demand an optimized approach for storing the problem and solving the problem in reasonable time. In addition, some of the problems formulate in a way that they exceed the existing computational limit. Also, they have a specific structure which varies for different problems and needs to be exploited in order to master the solving strategy. Here, structure refers to the pattern of the non-zero elements in the problem matrices.

3 Single-method Solver Systems

Solvers can be categorized as follows: direct and iterative. Direct solvers give an exact solution of the linear system. Direct solvers are a very general technique for solving sparse linear systems. They have high numerical accuracy and work even for sparse matrices with irregular patterns. Direct solvers perform a factorization of the original matrix A, into a product of multiple matrices, such as L, U. In the equation for linear systems: $Ax = b$ can be now written as $LUx = b$, where L and U are the factors of A. Although factors can be reused for multiple right-hand sides (rhs), direct solvers are costly in terms of memory for factors. However, this kind of approach is suitable for smaller problems, where an exact solution is possible because most likely, one solver is capable of solving the system after suitable selection of the solver method. But for very large sparse problems, an exact solution is not possible or desirable because of excessive run time and the kind of solution required changes from exact to approximation. This introduced the second class of solvers, called the iterative solvers, which use an initial guess to get an approximation of the solution. For a given approximation solution x_k, we assume that it is better than the x_{k-1} solution and keep updating the solution until we get close enough to the actual solution. Iterative solvers are capable of solving very large problems with or without the use of preconditioners. The efficiency of these solvers depends on the properties of the problem in hand and the preconditioning. These solvers do not involve matrix factors and instead involve matrix-vector products. This scheme is cheaper than the direct solver scheme because it uses less memory and fewer flops. However, solutions with multiple right hand sides can be problematic. In this section, we discuss in detail the direct solvers and the iterative solvers.

With respect to the number of solution methods used for a single linear system, there are two main solution approaches: single method and multiple methods. The traditional approach of solving linear systems is to choose a single optimized solver technique based on the dimension space and the physics of the problem, and to apply that solver method to obtain a solution. The second approach is by using more than one solver technique to obtain the solution. This section elaborates on the traditional approach and discusses some popular systems that use this technique.

In the single-method approach, only one method is used to solve the given linear system, as shown in [30, 59, 58, 44, 68, 76, 40, 66, 55, 78]. Based on the characteristics of the problem, the choice of solver is made. For instance, for symmetric positive definite matrix, Conjugate Gradient is a suitable choice. For non-symmetric matrices, BiCG becomes more preferable. For sparse least squares problems ill conditioned problems, Sparse QR factorization is a popular choice. For well-

conditioned problems, Cholesky factorization is used. If the method fails to solve the system, there is no other solving technique applied to the problem. Either of the two kinds of solvers can be used in this approach depending on the kind of problem. Direct solvers being the exact solutions and iterative being the approximation of the solutions can be used for small and large problems depending on the requirements. If an approximation of a solution is good enough, iterative solvers can be used. Although as the problems grow bigger iterative solvers become a preferable choice, however in some applications with bigger problem size also, direct solvers are used because of non-familiarity in iterative solutions. On the other hand, using multi-method schemes, makes it complex, as the number of decisions to be made are more, for instance, which all base methods to be used, when should a new solver be applied, which solver should be applied next, when do we eliminate a solver from the list of base methods, etc. Using a single solver scheme makes it easy, once the solver to be used is known, because the decision, once taken, is the only opportunity to decide which solver to pick. Numerical properties of a system can change during the course of nonlinear iterations and the solving scheme does not take that into consideration. This is a downside of using a single solver.

All the direct and iterative methods described in this section can be used as standalone solver methods for solving sparse linear systems.

3.1 Direct Solvers

For linear systems $Ax = b$, where A is the coefficient matrix (sparse), x is the solution vector, b is the right-hand side vector (known vector) sometimes with all elements set to one, direct solvers [30, 25] provide an exact solution, $x = A^{-1}b$, when A is invertible, for the linear system in a finite number of steps, and they are more robust than iterative solvers. These solvers work very well with small matrices and when exact solutions are required. However, they don't work well with very large matrices, because they may be too expensive. The time complexity of direct solvers can be given as, $O(n^3)$ and the space complexity is generally $O(n^2)$. We focus on large sparse problems for our research, and for such matrices, direct solvers may not necessarily be the optimal choice for solving these systems. This is because the memory requirement for direct solvers can be huge, for an $n \times n$ matrix. The reason is, the program requires to store $n \times n$ elements of matrix A, whereas sparse systems only require non-zero elements to be stored, which means it needs storage of $3 * nnz(A)$, where nnz is the number of non-zeros and for each non-zero entry it needs to store the row and column number it belongs to and the value of the element, making it three values for each non-zero entry. In addition, direct methods cause fill-in, which introduces additional non-zeros during the factorization process, which can be very large. There have been modifications made to these kinds of solvers to work with sparse matrices; however for the scope of this work we consider using iterative solvers for now. In this section, I briefly describe some of the most commonly used direct solvers.

3.1.1 LU Factorization

The LU method is a technique in which an $n \times n$ matrix A is factored into a product of lower triangle and upper triangle of the matrix and shown by the equation: $A = LU$ where A is the matrix, with elements $a_{i,j}$, with i rows and j columns; L is the lower triangle and U is the upper triangle matrices. An $n \times n$ square matrix L with elements l_{ij} is called lower triangular if $l_{ij} = 0$ for $i \leq j$, i.e., all elements above the diagonal are zero. An $n \times n$ square matrix U with elements u_{ij} is called Upper Triangular if $u_{ij} = 0$ for $i \geq j$ i.e. all elements below the diagonal are zero.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}$$

Figure 7: LU factorization for matrix A.

The LU Factorization of A having elements, a_{ij} is given as the product of these two matrices L and U, where $l_{i,j} = 1$. L and U are obtained by applying various row operations to make all the elements below the diagonal and all the elements above the diagonal as zero, respectively. The LU factorization is shown in Figure 7. This is then followed by forward and backward substitution to get the solution. The total time is dominated by decomposition, which is $O(1/3n^3)$. Forward and backward substitution is shown as below:

Forward substitution for lower triangular system $Ax = b$

$$x_1 = b_1/a_{11}\ x_i = [b_i - \sum_{j=1}^{i-1} a_{ij}x_u]/a_{ii}, \text{ where } i = 2, \ldots, n.$$

Backward substitution for upper triangular system $Ax = b$

$$x_n = b_n/a_{nn}\ x_i = [b_i - \sum_{j=i+1}^{n} a_{ij}x_j]/a_{ii}, \text{ where } i = n-1, \ldots, 1.$$

This method leads to a unique and robust solution. However the disadvantage of this scheme is the large memory requirement because of fillin during factorization. In the process of fillin, zeros are converted to non-zeros, which makes the number of entries substantially more for the factors than there were for the original matrix, thus increasing the memory requirement. The time complexity for LU Factorization is $O(2/3n^3)$ and its space complexity is $O(n^2)$. One additional feature intrinsic to this strategy is that, if the diagonal elements are zero in the start or in any of the intermediate stages, this method will fail. In order to address this issue, it becomes mandatory to use a preconditioner with this method.

$$\begin{aligned} A &= \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \\ &= \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} . \begin{bmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{bmatrix} = LL^T \\ &= \begin{bmatrix} l_{11}^2 & l_{21}l_{11} & l_{31}l_{11} \\ l_{21}l_{11} & l_{21}^2 l_{22}^2 & l_{31}l_{21} + l_{32}l_2 \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{bmatrix} \end{aligned}$$

Figure 8: Cholesky method for matrix A.

3.1.2 QR Factorization

LU factorization and the Cholesky method are based on Gaussian elimination, whereas QR factorization is an orthogonalization method. Some problems cannot be solved with Gaussian elimination, as it does not preserve the Euclidean norm, which therefore does not preserve the solution to the problem. In such situations QR factorization is applicable. In QR factorization [39], the matrix is factorized into the product of two matrices: Q, the orthogonal matrix, and R, the upper triangular matrix; i.e.,$A = QR$. The orthogonal matrix is a matrix such that the product of the orthogonal matrix and its transpose give the identity matrix,

$$Q^T Q = I \text{ and } Q^{-1} = Q^T.$$
$$A = QR \text{ where } R = Q^T A$$

Now instead of solving $Ax = b$, $Qx = b$ is solved by simply computing $Rx = Q^T b$. QR factorization is one of the simpler methods which converts the problem into a triangular problem that is easier to be solved by forward or backward substitution. So similar to Gaussian elimination methods, this method also tries to introduce zeros in order to make the problem in the Upper Triangular format. QR factorization can be computed by many ways, such as plane rotation, Householder transformation and Givens rotation. One popular way is using Gram-Schmidt orthogonalization, which is explained in detail below. There are three main steps of the QR factorization:

1. Find the orthogonal basis for the problem using Gram-Schmidt method: It orthonormalizes a set of vectors in an inner product space. It takes $a_1, a_2, \ldots, a_k$ and generates an orthogonal set $u_1, u_2, \ldots, u_k$ where a_k are the columns of the original matrix, A . The orthogonal basis is computed by the formula shown below:

$$u_k = a_k - \sum_{j=1}^{k-1} \text{proj}_{u_j} \text{a}_k$$

2. Convert the orthogonal basis into orthonormal basis: This conversion is done to make them of uniform length. This is computed as follows:

$$e_k = u_k / ||a_k||$$

   Here e_k is the normalized vector and $||a_k||$ is the length of vector a.

3. Perform the QR factorization: Once the first two steps have been performed, this will give Q, which is the normalized vector e_k obtained in Step 2. R is obtained by applying the formula $R = Q^T A$.

Note that this method requires separate storage for A, Q and R matrices because A is used for the inner loop calculations and hence cannot be discarded. The space complexity of this method is $O(n^2)$ and the time complexity is $O(4/3n^3)$. Modified Gram-Schmidt method can be applied to address this high storage requirement by making A and Q share the same storage. Therefore, orthogonalization methods are more expensive than Gaussian elimination methods, for instance Cholesky method.

3.1.3 Cholesky Method

The Cholesky method is a popular direct method for symmetric positive definite matrices, and the factorization can be shown by the equation: $A = LL^T$ where L is a lower triangle matrix with positive entries on its diagonal. The factorization is shown in Figure 8. New nonzero entries that appear in A are called fill-in. The system can be solved by computing $A = LL^T$, followed by solving $Ly = b$, and later solving $L^T x = y$. This method involves computing square roots of some of the elements, such as the first element in the first row. This method is very popular for its efficiency and stability when solving symmetric linear systems, for the following reasons. Cholesky only requires the lower triangle of the matrix to be stored, so the upper triangle need not be stored at any point in time. It requires only $(n^3)/6$ multiplications and a similar number of additions. This is half of the storage requirement and half of the operations required by other direct methods for nonsymmetric systems, such as LU factorization. Computing square roots requires positive entries, which ensures that the algorithm is well defined. In addition, this method does not require pivoting of any form for stability purposes. In contrast to LU decomposition, Cholesky is more efficient with its time complexity better by a factor of 2; that is, $O(1/3n)^3$. The space complexity of this method is $O(n^2)$.

Cholesky can be used for problems other than positive definite with a variation (the original Cholesky method fails if the problem involves negative values, as it requires taking square root of a negative element). This problem can be avoided by using a variant of the Cholesky method, which factorizes the system as follows: LDL^T factorization where D is the diagonal matrix of the squares of the diagonal entries. This ensures that this variant does not require square roots of any elements. This variant is also referred to as the Augmented System method. It is represented as follows:

$$\begin{bmatrix} I & A \\ \mathrm{A}^T & 0 \end{bmatrix} \cdot \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

The matrix above is symmetric, positive definite which can be solved with this factorization. If the matrix is ill conditioned, then I can be replaced with αI for improvement because LDL^T factorization is not very efficient with ill conditioned A.

3.1.4 Frontal Solver Method

Frontal solver method [48] is a method for solving sparse linear systems which are symmetric positive-definite and banded. They are very popular in finite element analysis. It is a slight improvement of the Gaussian elimination and performs better by eliminating more finite elements. Each finite element is associated with some variables. This method declares a frontal matrix, which is a dense square sub-matrix, in which all the operations are performed. It starts eliminating finite elements and moves downwards in a diagonal fashion, element by element. It assembles the finite elements (based on an order defined prior to the assembly), then eliminates and updates variables, and then again does the assembly. This alternate cycle keeps going till the frontal matrix gets filled. This process is shown in Figure 9 At this stage, the frontal matrix consumes the maximum memory, and from here on, the frontal matrix does not grow in size. Once the frontal matrix is full, a partial factorization is applied on the frontal matrix, and elimination is performed. The elements that are fully summed are eliminated, and all the others elements are updated. A variable is fully summed when the last equation in which it occurs is assembled. Those elements that are selected for elimination are removed from the frontal matrix and placed elsewhere. This is followed by assembly of the new finite elements, which earlier could not be assembled because the frontal

matrix had become full. This process continues until all the elements have not been assembled and all variables have been eliminated. The next step is to solve the linear system, using forward and backward substitution.

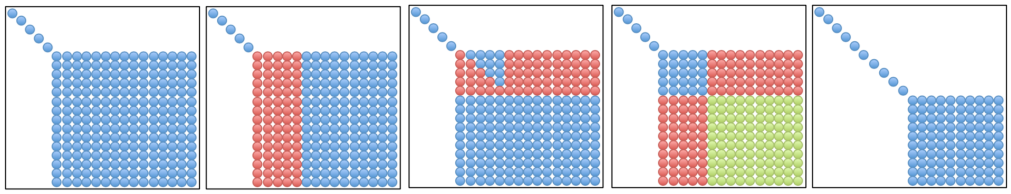

Figure 9: Standard LU factorization for matrix A.

3.2 Iterative Solvers

Iterative solvers [37, 67, 77, 5, 4] start with an initial guess for the solution and successively improve until the solution is accurate enough and is acceptable as a solution. The time complexity for iterative solvers is $O(n^2)$ per iteration. One popular class of iterative numerical solvers is the family of Krylov subspace methods. This class of solvers decomposes the solution space into several subspaces, which are simpler than the original subspace. These methods form a sequence, called the Krylov sequence, shown below:

$K_k(A, r_0) = span\{r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0\}$

where k is the order of the subspace, A is an $n \times n$ matrix, v is a vector of dimension n, r_0 is an initial vector of successive matrix powers times the initial residual (the Krylov sequence). The subspace is the successive powers of the matrix A starting from 0 to $k-1$ applied to the residual form. Minimizing the residual over the subspace formed then forms the approximations to the solution. The main operations in a Krylov subspace method are (i) matrix-vector products, (ii) dot products and norm computations, and (iii) vector updates. They are among some of the successful methods for solving linear and non-linear systems because of their efficiency and reliability. Iterative solvers provide an approximation of the solution, as the exact solution might be too expensive to compute or may not exist. Iterative solvers start with an initial guess and generate successive approximations to the solution. In cases of large linear systems, iterative methods are more useful. The traditional approach of solving large sparse linear systems involves using a solver combined with a preconditioner. There are many solver techniques that exist for solving large sparse linear systems of the form: $Ax = b$. The residual norm can be given as $||Ax - b||$. The aim with iterative solvers is to reduce the residual norm as much as possible.

These methods are preferable than the direct solver methods for many reasons. First, iterative methods are more parallelizable in respect to scalability than direct methods because iterative methods use matrix-vector products instead of matrix-matrix products. Matrix-matrix multiplication operations are more expensive computationally due to more floating-point operations required per memory access. Also, it is harder to program a scalable matrix-matrix multiplication implementation. As the number of processors increase for a given problems, the communication overhead increases. Second, iterative methods require less storage space because they do not perform a fill-in operation for the coefficient matrix. This makes them less expensive than the direct methods

which give an exact solution. Also if the initial guess is good for iterative solvers, the problem can converge in very few steps.

Further in this section, I list some of the most popular Krylov subspace methods for symmetric and non-symmetric matrices.

3.2.1 Conjugate Gradient Method

Conjugate Gradient (CG) method [44, 60, 56] is applicable for symmetric linear systems. CG starts with an initial guess of the solution, an initial residual and an initial search direction. It looks for approximate solutions at each step within its Krylov subspace as shown below:

$$K_k = span\{b, Ab, A^2b, \ldots, A^{k-1}b\} \text{ for } k \geq 1$$

It finds a series of gradients (conjugate vectors) $x^0, x^1, \ldots$ until the point where the gradient gets close enough to the solution. The gradients are given by the equation shown below:

$$x_{k+1} = x_k + \alpha_k s_k$$

where α_k is a scalar determining the step length and s_k is the search direction.

The minimum over α occurs when the residual is orthogonal to the search direction. Each iteration requires only one matrix-vector multiplication. They are also the gradients of a quadratic function shown below, the minimization of which is the same as solving the system

$$\phi(x) = 1/2x^T Ax - x^T b.$$

Initially, $s_0 = r_0 = b - ax_0$, and the residual is updated in the following way:

$$r_{k+1} = b - Ax_{k+1} = r_k - \alpha_k A s_k.$$

The storage requirement of this method is low, as it only needs to store vectors x, r and s, and they can be overwritten after each iteration. The convergence rate depends on the square root of the condition number. It is an extremely popular solver technique for symmetric, positive definite matrices. Conjugate Gradient method became the ground for many other solver methods, which were variants of CG, such as BiConjugate Gradient method, BiConjugate Gradient Stabilized method, Conjugate Gradient Squared and Improved Stabilized version of BiConjugate Gradient Squared method (some of which are described in the following section).

3.2.2 Variants of Conjugate Gradient Method

The Biconjugate Gradient(BiCG) method [78] is useful for non-symmetric and non-singular matrices. Its implementation is similar to running the Conjugate Gradient except that it maintains a second Krylov subspace, so there are two sequences of vectors, one based on A and other on the transpose of the matrix A^T. At each iteration, it requires a multiplication of A and A_T, therefore BiCG has twice as many matrix-vector multiplications and dot products than Conjugate Gradient method. It solves the system $Ax = b$ as well as $A^T x^* = b^*$. The two sequences of residuals that are updated in BiCG can be seen below:

$$r^{(i)} = r^{(i-1)} - \alpha_i A x^{(i)}$$
$$r^{*(i)} = r^{*(i-1)} - \alpha_i A^T x^{*(i)}$$
where
$$\alpha_i = {r^{*(i-1)}}^T r^{i-1} / {x^{*(i)}}^T A x^i$$

The advantage of BiCG over Conjugate Gradient method is that it can solve a wider variety of matrices, such as symmetric positive definite matrices and non-symmetric matrices.

The Biconjugate gradient stabilized method (BiCGStab) [76] is an iterative Krylov subspace method for the numerical solution of non-symmetric linear systems which is a variant of the Biconjugate gradient method (BiCG). It computes $i|- > Q_i(A)P_i(A)r^{(0)}$ where Q is an ith degree polynomial describing a steepest-descent update. It has a faster convergence than the original method, as well as other variants such as the conjugate gradient squared method (CGS). This method involves two inner products more than the BiCG method. The advantage is, it avoids the irregular convergence patterns, which are observed in other variants of Conjugate Gradient method.

Improved Stabilized version of BiConjugate Gradient Stabilized (IBiCGStab) method [54] is another variant of Conjugate Gradient method series. It is an improved stabilized version of Bi-Conjugate Gradient Stabilized method. This method attempts to reduce the computational cost of the BiCGStab method.

The Conjugate-Gradient Squared (CGS) [72] method is another variant of the BiCG algorithm. The difference between them is that CGS does not involve adjoint matrix-vector multiplications, and the convergence rate is expected to be better than the BiCG method.

3.2.3 LSQR Method

LSQR [58, 59] is an algorithm for non-symmetric sparse linear equations. It is one of the oldest methods in the history of iterative solvers and is similar to Conjugate Gradient method. This method undergoes a bidiagonalization procedure and generates a sequence of approximations in a manner that the residual norm, $||r_k||$ decreases monotonically. Given an initial guess and a starting vector b, it uses the Lancoz process to reduce the problem to a tridiagonal form. The Lancoz method generates a sequence of vectors v_1, w_1 which help convert the original problem matrix to tridiagonal form shown below:

$$\begin{gathered}\beta_1 v_1 = b \\ w_1 = Av_1 - \beta_i v_{i-1} \\ \alpha_i = v_i^T w_i \\ \beta_{i+1} v_{i+1} = w_i - \alpha_i v_i \text{ for } i = 1, 2, \ldots, k\end{gathered}$$

After k steps, it becomes:

$$AV_k = V_k T_k + \beta_{k+1} v_{k+1} e_k^T.$$

Here, T_k is the $tridiagonal(\beta_i, \alpha_i, \beta_{i+1})$ and $V_k = [v_1, v_2, \ldots, v_k]$ and α, β are the scalars generated by the Lancoz method. Now for solving the system where x_k is the sequence of vectors, the following equations are multiplied by an arbitrary vector y_k whose last element is η_k.

$$\begin{gathered}T_k y_k = \beta_1 e_1 \\ x_k = V_k y_k\end{gathered}$$

Once the $\eta_k \beta_{k+1}$ is negligibly small, it becomes the termination criterion for the method.
Some of the advantages of this solver are high efficiency and high convergence speed for a variety of linear systems. The memory requirement for these systems scales with the dimensions of the matrix and work well in parallel setups, as the matrix A is used only for computing products of the form Av and $A^T u$. It has been observed in the past that this method is more reliable than the other solver techniques and hence is popular in many domains; for instance, it has been a common choice for seismic tomography since many years in the past.

3.2.4 GMRES Method and Its Variants

This category of solvers includes the Generalized minimal residual (GMRES) and variants of it, such as, Flexible GMRES and LGMRES. GMRES method [68, 49] and its variants are used for non-symmetric matrices. They approximate the solution by generating a sequence of orthogonal vectors for a Krylov subspace. MINRES [57] is similar to Conjugate Gradient and applies the same solving scheme. Since Conjugate Gradient is only for symmetric definite matrices and MINRES is applicable for symmetric indefinite matrices. GMRES is similar to MINRES because it also computes a sequence of orthogonal vectors because the Arnoldi method is used to find the orthogonal vectors and it produces an upper Hessenberg matrix using the Gram Schmidt method. The GMRES method at each step minimises the norm of the residual vector over a Krylov subspace. Below are the steps followed in GMRES method:

1. Choose the initial guess to begin, x_0 and compute the residual, $r_0 = b - Ax_0$ and $v_1 = r_0/||r_0||$.
2. For each iteration, calculate the orthogonal vectors with Arnoldi method and minimize the residual over $x_0 + K_k$, where K_k is the k^{th} Krylov subspace, and then minimize the residual as follows:

$$||b - Ax_k|| = \min_{X \epsilon X_0 + K_k} ||b - Ax|| \; h_{i,j} = (Av_j, v_i), i = 1, 2, \ldots, j$$
$$v'_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_i$$
$$h_{j+1,j} = ||v_{j+1}||$$
$$v_{j+1} = v'_{j+1}/h_{j+1,j}$$

3. Set the approximate solution as:

$$x_k = x_0 + V_k y_k, \text{ where } y_k \text{ minimizes the residual.}$$

4. Repeat until the residual is small enough.

The number of multiplications required by this method is $\frac{1}{2}k^2 n$, where k is the number of iterations and n is the number of rows. The storage requirement for GMRES is more than MINRES and Conjugate Gradient because it needs to store the entire sequence of vectors (all the successive residual vectors) for the Krylov subspace. The GMRES method is restarted after a certain amount of vectors have been generated to reduce the memory requirement. Choosing the number of vectors after which the method is restarted plays a crucial role in deciding whether the method will converge. If a very small k is used, it may result in poor or no convergence.

FGMRES method [66, 55] is a generalization of GMRES that allows greater flexibility in the choice of solution subspace than GMRES method by allowing any iterative method to be used as a preconditioner. LGMRES [7, 6] augments the standard GMRES approximation space with error approximations from previous restart cycles. This method supersedes the original method by accelerating the convergence of restarted GMRES.

3.2.5 Quasi-Minimal Residual Method (QMR)

QMR [38] is a quasi-minimal residual method for non-symmetric linear systems and is an improvement over the BiCG method. QMR can also be compared to GMRES and one can say that

the former is better because GMRES usually converges too slowly for difficult problems because of the restarts that GMRES needs. QMR overcomes the problems of BiCG breakdown and slow convergence of GMRES. It looks ahead using the Lancoz Algorithm to generate vectors for the Krylov subspace.
It uses two non-zero starting vectors, v_1 and w_1, and generate two sequences of vectors given as $v_1, v_2, \ldots, v_n$and $w_1, w_2, \ldots, w_n$ such that

$$span(\mathrm{v}_1, v_2, \ldots, v_n) = \mathrm{K}_n(v_1, A)$$
$$span(\mathrm{w}_1, w_2, \ldots, w_n) = \mathrm{K}_n(w_1, A^T)$$

If $w_{n+1}^T v_{n+1} = 0$, the Lancoz algorithm has to be terminated because this would make the system fail in the next iteration as it will introduce division by zero case. This can happen in three ways: (1) $w_{n+1}^T = 0$ (2) $w_{n+1} = 0$ or (3) $w_{n+1}^T \neq 0$ and $v_{n+1} \neq 0$. This is called serious breakdown. This is where the QMR uses look-ahead to make the decision to terminate, thus avoiding the breakdown condition. Given an initial guess, x_0 and r_n gives the residual vector r_n

$$r_n = b - Ax_n \text{ and } x_n \in x_0 + K_n(r_0, A)$$
$$x_n = x_0 + V^{(n)z} \text{ where } V \text{ is the block generated from the sequence } v_1, v_2, \ldots, v_n$$
$$AV^{(n)} = V^{(n+1)}H^(n)_e$$

where H is the upper Hessenberg matrix which can be reduced to tridiagonal matrix using a diagonal matrix represented by $\omega^{(n)}$, which gives the QR decomposition as shown below:

$$\omega^{(n)} H_e^{(n)} = Q^{(n)H} A = \begin{bmatrix} R^{(n)} \\ 0 \end{bmatrix}$$

Once the QR factorization is updated, the residual is computed as follows, where $t^{(n)}$ is obtained from z^n :

$$x_n = x_0 + V^{(n)}(R^{(n)})^{-1} t^{(n)}$$
$$z^n = (R^{(n)})^{-1} t^{(n)}$$

QMR is similar to the BiCG method, and takes almost same number of iterations to solve as BiCG except, that it is more robust than BiCG because of the lookahead strategy described above. Due to this QMR has smooth convergence curves and good numerical stability. When compared to BiCG, QMR is less prone to breakdowns and is more stable. It updates the Euclidean norm instead of the norm; hence, the word Quasi which means "almost". TCQMR [31] is a variant of QMR provided by Tony Chan with improvement over the QMR Method which avoids the matrix-vector multiplications with A^T.

3.2.6 Chebyshev Method

The Chebyshev method [40, 41] can be used for symmetric, positive definite or non-symmetric sparse linear systems. It is a variant of the Conjugate Gradient method, except that Chebyshev avoids computing inner products, removing the operations with the inner products completely, unlike GMRES, Conjugate Gradient and other orthogonalization methods whose parallel implementations require communication intensive inner products. The only inner products performed by the Chebyshev method are the inner products required for monitoring of convergence, which can be done only occasionally and not for every iteration. The occasional convergence check is due to the

special property of this method, which is a reliable forecast of the convergence rate. This method needs knowledge about the spectrum of the matrix A, given by the lower estimate of the smallest Eigenvalue and the upper estimate of the largest Eigenvalue. Once the Eigenvalues are known, the iteration parameters become known as well and the ellipse enveloping the Eigenvalues is identified. Two scalar values, c, d are needed to define the ellipses such that they have common center $d > 0$ and foci $d + c$ and $d - c$ which contain the ellipse that surrounds the spectrum. Let the center of the ellipse be α and so the foci are given as $\alpha \pm c$.
The Chebyshev method can be defined by translating the Chebyshev polynomial, T_n from the interval [-1,1] to the interval I, which is followed by scaling which makes their value at 0 to be 1.

$$\mathrm{T}_n(I) = \begin{cases} cos(narcos(I)) & if|I| \leq 1, \\ cosh(narcosh(I)) & if|I| \geq 1, \\ (-1)^n cosh(narcosh(-I)) & if|I| \leq -1 \end{cases}$$

The residual polynomial,p_n that characterizes the Chebyshev iteration is as follows:

$$p_n(I) = (T_n((I - \alpha)/c))/T_n((-\alpha)/c$$

Given the initial guess of the solution x_0 and the residual to be $r_0 = b - Ax_0$, then the nth approximation of the solution and residual should satisfy:

$$b - Ax_n = r_n = p_n(A)r_0.$$

The Chebyshev iteration obtained is optimal, for each iteration it generates the smallest maximum residual. These computed residuals do not cause any slowdown when compared to BiCG and and Conjugate Gradient method, nor do they have higher computational cost; in fact, the overall cost is lower, since it skips computing the inner products completely except the ones that are inexpensive.

3.2.7 Jacobi Method

Jacobi method is a solver in which the original matrix A is split into two matrices, say, S and T, such that, $A = S + T$, where S is the diagonal matrix of A and T is the original matrix with the diagonal matrix removed, leaving the remainder, shown as,

$$S = diag(A) = D_A \text{ and } T = A - D_A$$

Once the diagonal matrix is obtained and the original matrix is split into S and T matrices, an update is performed according to the following update rule:

$$Ax = b,\ A = S + T \text{ and } S = D_A$$
$$(S + T)x = b$$
$$(D_A + A - D_A)x = b$$

For each iteration, to obtain the next successive solution the below rule is used.

$$x^{(k+1)} = D_A^{-1}(b - (A - D_A)x^k)$$

This method has a simple formulation; given an initial guess, let us say x^0, for the solution, we can obtain x^k so that

$$x_i^k = [b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{k-1}]/a_{ii}.$$

This method requires double storage for the solution vector, because all the old values are needed for the sweep process to get the new values; so at a given time, it stores the old and the new values of x shown in equation above, and that causes the double storage demand. In addition, the Jacobi method is extremely slow, because in each iteration, it solves every variable locally with respect to other variables. So each variable is solved once in each iteration. The rate of convergence is $(\pi^2/log10)s^2/2$. Here s is the mesh size, which is $1/n + 1$, for an $n \times n$ matrix. This is a solver method that is common for diagonally dominant system. One of the reasons it is very popular is because it is highly parallelizable.

3.2.8 Gauss-Seidel Method

This method is a popular technique for diagonally dominant, symmetric and positive definite matrices. Gauss-Seidel method is similar to the Jacobi method and uses the same scheme of solving, except that this method takes advantage of the fact that the latest information obtained in the first iteration can be used in the subsequent iterations. The formula is

$$x_i^k = (b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k-1} - \sum_{j=i+1}^{n} a_{ij} x_j^k)/a_{ii}.$$

This method converges faster than the Jacobi method. Also, the computation cost per iteration is lower than Jacobi and Conjugate-Gradient because it applies the updated values of the variable as soon as they become available during the iteration. Therefore it is cheaper in terms of cost per iteration. This method also has an added advantage of using the updated values right away; it does not require duplicate storage for the solution vector. Although the convergence rate is better for this method than Jacobi method by a factor of two, it is still slow. The rate of convergence is $(\pi^2/log10)s^2$ where s is the mesh size, which is $1/n + 1$, for an $n \times n$ matrix.

However, the disadvantage of the GS method is, with this scheme, although it allows using the latest information obtained during the present iteration to be used in the upcoming iterations, it loses the parallelizability. Parallelism is discussed in detail in Section 5. In addition, Jacobi and Gauss-Seidel method remove the high frequencies of the error very quickly, however the iteration stops if the error is a smooth function. This causes slow convergence to the solution for both these methods.

3.2.9 Successive Over-Relaxation Method

Successive Over-Relaxation (SOR) [42] is an improvement of the Gauss-Seidel method. Although the Gauss-Seidel Method is faster than the Jacobi Method, it is still slow. SOR achieves an improvement over the convergence rate of the Gauss-Seidel method by reducing the norm of the residual vector. Starting with x_i^k, x_i^{k+1} is computed the way Gauss-Seidel method would compute, shown by the equation below:

$$x_i^{k+1} = (b_i - \sum_{j=1}^{i-1} a_{ij} x_j^k - \sum_{j=i+1}^{n} a_{ij} x_j^{k+1})/a_{ii}.$$

Also written as $x_i^{k+1} = x_i^k + r_{ii}^{k+1} a_{ii}$ where r is the residual vector.

SOR uses the next iteration as a search direction with a fixed relaxation parameter, called ω. So the above equation changes to

$$x_i^{k+1} = x_i^k + \omega r_{ii}^{k+1} a_{ii}$$

That is, x_i^{k+1} is computed by taking a weighted average of the present iteration and the next Gauss-Seidel iteration. The value of ω, if taken as one, gives the Gauss-Seidel method. A value less than one gives under-relaxation, which takes more time to converge than the Gauss-Seidel method. A value in between one and two gives over-relaxation and gives a faster convergence rate than the Gauss-Seidel method. The rate of convergence for SOR is much improved from Jacobi and Gauss-Seidel. The convergence rate is $(2\pi/log10)s$; where s is the mesh size which is $1/n+1$, for an $n \times n$ matrix.

3.3 Comparison of Direct and Iterative Solvers

Direct solvers solve the system in a finite number of steps and provide an exact solution for the system. These methods have high numerical accuracy and are reliable, as the solution does not depend on the characteristics of the problem. These solvers do not generate a solution until the entire solving process has finished. Such solvers are suitable for small systems. However, for large problems, they become very expensive, and in some cases, direct solvers cannot solve the system because of the high memory requirement. It requires $O(n)^3$ operations for solving the system and the memory requirement is also $O(n)^3$. Therefore the direct solvers do not scale very well for larger problems.

Iterative solvers, on the other hand, are approximations of the solutions. They start with an initial guess of the solution and proceed until they reach close enough to the actual solution and achieve desired accuracy. Each iteration improves accuracy of the solution, and the solving process stops once the estimated error is equal or below the acceptable tolerance. The efficiency of these methods depends on the type of problem, and the convergence rate depends on the condition number of the system and preconditioning. It is hard to predict the number of iterations it takes to get to the solution for these solvers. The speed of the solver depends on the number of non-zero elements present in the system. It is less expensive in terms of memory; the memory requirement is $O(n)$, where n is number of non-zero elements and typical convergence is less than $O(n)$ iterations. The storage requirement is shown in Table 3.3, and operations required per iteration are shown in Table 3.3, for all iterative solvers discussed in this report. In the table, i refers to the iteration number, n is the number of rows and nnz are the number of non-zeros. These solvers are less reliable, as they may not be able to converge to a solution. There can be multiple reasons for a solver to not converge, such as, the time it takes to get to a solution may be very high or, it may not have been solved accurately enough to be accepted as the solution. Iterative solvers are more suitable for large sparse linear systems.

Solver	Storage requirement
QMR	nnz + 16 n
Chebyshev	nnz + 5 n
GMRES	nnz + (i + 5)n
Conjugate Gradient	nnz + 6 n
BiCG	nnz + 10 n
BiCGSTAB	nnz + 10 n
CGSTAB	nnz + 11 n
Jacobi	nnz + 3 n
SOR	nnz + 2 n
LSQR	nnz + 2 n
Gauss-Seidel	nnz + 2 n

Table 1: Storage requirement of iterative solvers.

Solver	Matrix-Vector Products	Inner Products
QMR	2 (1 with A, A^T)	2
Chebyshev	1 (with A)	0
GMRES	1 (with A)	i+1
Conjugate Gradient	1 (with A)	2
BiCG	2 (1 with A, A^T)	2
BiCGSTAB	2 (with A)	4
CGSTAB	2 (with A)	2
Jacobi	1 (with A)	0
SOR	1 (with A)	0
Gauss-Seidel	1 (with A)	0
LSQR	2 (with A)	2

Table 2: Operations per iteration for iterative solvers.

Although iterative solvers can get to a solution faster, direct solvers still tend to remain useful, as in many cases iterative solvers cannot produce a solution. For instance, scientists in the domain of nuclear physics continue to use a direct solver (SuperLU), even though iterative solvers exist. The reason is that using iterative solvers cause the system to fail in finding solution. The overall aim is to deliver solutions for problems in the best time without frequently encountering scenarios in which some problems have no solutions. Therefore, both direct solvers and iterative solvers continue to exist, as each solve different kinds of problems and techniques have been developed that use multiple solvers to improve both efficiency and robustness. These techniques will be discussed in detail in the later sections of this report.

3.4 Accuracy of Solutions

Once a solution has been obtained. The solution can be validated using different metrics. In this section we discuss two of the popular metrics.

1. **Residual of a solution**: In order to check the validity of a solution, the easiest way is to plug it in the equation and compare how close the left and right sides of the equation are to each other. The residual vector of a computed solution x' for the linear system $Ax = b$ can be given as:

$$r = b - Ax^{'}$$

A large residual implies a large error in the solution. For direct solutions, we desire the error E to be equal to zero, which is given by the equation shown below.

$$E = ||x' - x|| = 0$$

For iterative solutions, the computed solution is an approximation of the actual solution, so we also want the error to be as close to zero as possible. However, a small residual does not necessarily mean it is a good indication as the computed solution is close to the true solution

because some methods always produce a small residual regardless of the system characteristics. An example of such a method is Gaussian elimination with backward substitution.

2. **Estimation with condition number:** Conditioning is a characteristic of a system given by the formula $cond(A) = |A|.|A^{-1}|$. The condition number can determine the possible relative change in the solution for relative changes in the entries of the matrix. Therefore it can give an estimate of the error in the computed solution. In other words, changes in the input, i.e., A and b of the equation $Ax = b$, get multiplied by the condition number to produce changes in the output, i.e. x in $Ax = b$. This means, small errors in the input operations can cause large errors in the solution of the system. Hence a large value for condition number for a matrix means it is ill conditioned. A smaller value implies a well-conditioned matrix.

4 Preconditioning

The general strategy of solving a linear system involves transforming the system into another system which has the same solution x and that is easier to be solved. This process is known as Preconditioning [10]. This is done by combining a solver method with a preconditioner. One such transformation is pre-multiplying a linear system with a non-singular matrix. In other words, multiplying the left-hand side and the right-hand side with a non-singular matrix P.This process leaves the system unaffected. This transformation is shown below:

$$Ax = b$$
$$PAx = Pb$$
$$x = (PA)^{-1}Pb = A^{-1}P^{-1}Pb = A^{-1}b$$

Another transformation is by introducing the product of a matrix, P and its inverse P^{-1} in the original system. Since a matrix when multiplied by itself results in identity matrix, it has no affect on the system.

$$Ax = b$$
$$(PP^{-1})Ax = b$$
$$P^{-1}APx = b$$
Introducing y where $y = Px$ and substituting in above equation gives:
$$P^{-1}Ay = b$$

So the system of equation changes from $Ax = b$, to $P^{-1}Ay = b$, and $Px = y$. The transformed system can be more easily solved because of the change in the condition number. The original matrix A, had a higher condition number than the transformed system $P^{-1}A$. A system with a higher condition number is more ill conditioned than a system with a lower condition number. The convergence rate of iterative solvers increases with a decrease in condition number. Therefore, preconditioning helps by improving the convergence rate of solvers and lets them get to the solution faster. Below is a list of popular preconditioners:

- Incomplete factorization (ILU): ILU is an approximation of the LU (Lower Upper) factorization. LU factorization factors a matrix as the product of the lower and the upper triangular matrix.

- Jacobi or diagonal: One of the simplest forms of preconditioning, in which the preconditioner is the diagonal of the original matrix.
- Additive Schwarz method: Solves an equation approximately by dividing the problem into smaller problems and adding the results to get the final result.
- Block Jacobi: It is similar to Jacobi, except that in this case, instead of the diagonal, the block-diagonal is chosen as the preconditioner.

5 Parallelism Issues

At the time that many of these solvers methods were developed, the main goal was to solve the problems at hand in the best time. Scaling these solvers was not a concern until they began to be used for new problems and the problem sizes exceeded the limit, and the efficiency of these methods became questionable.

Scaling linear systems depends on many factors. For instance, the amount of time will the solver takes to complete in a parallel environment. This time begins when the first processor begins and finishes when the last processor finishes. The second factor is how much better will the solver perform in the parallel environment and how does that compare to the serial version of the solver i.e., the efficiency (E) of the solver, which can be computed as follows:

$$E = S/f = T_s/fT_p$$

Here f is the factor of speedup, and T_s is the serial time and T_p is the parallel time.

With an increase in the number of processors, there can be a big communication overhead involved if the algorithm involves a lot of communication between different processors, as more information has to be transferred to multiple processors. The efficiency of the system can be maintained by maintaining the ratio T_0/W. The parallel time (T_p) can be given as:

$$T_p = (W + T_0(W, f))/f$$

Here W is the number of operations in the solving method and T_0 is the overhead time. The speedup and efficiency now become:

$$S = W/T_p = W_p/(W + T_0(W, f))$$
$$E = S/f = W/(W + T_0(W, f)) = 1/(1 + T_0(W, f)/W)$$

The need for scaling the existing solver methods has been growing with the prevalence and increasing scales of parallel architectures. For direct solvers, memory consumption is very high, because the direct solvers involve fill-ins, which are unknown and can cause load imbalance. Therefore designing a parallel algorithm that is efficient has proven to be challenging. In addition, direct solvers involve matrix-matrix products, which are more expensive computationally. On the other hand, although iterative solvers replace that with matrix-vector products; if iterative methods were to be performed in a parallel environment, the chances that the efficiency of the solvers would be good are low. This is because the way iterative methods work is, they perform an iteration and compute some quantities, like the norm of the matrix to check whether they have reached the desired approximation for the solution. This computed value has to be sent to all the processors, which requires global communication and global synchronization, as they have to be updated at the same time. In addition, each iterative solver suffers from scaling difficulties for more than 10,000 cores,

as each iteration requires a minimum of one vector inner product. Every inner product requires global synchronization and that does not scale well, due to internode latency.

One strategy for handling this situation is to avoid computation of these quantities that involve global communication for convergence check. The norm value is updated to the neighboring processors and propagated later. However, this, again, involves global communication and it also requires a complex decision regarding until what point the processor should be deferred from updating.

Multigrid methods [46] on the other hand, avoid these problems completely by solving a subsystem separately and avoid global communication completely. The mechanism of these methods is explained in detail below, followed by a description of one of the popular multigrid methods. Another approach that came into existence because of the scalability issues is the hierarchical approach of using solvers. Hierarchical solvers are also explained in the later part of this section.

5.1 Multigrid Methods

There have been many efficient methods designed for solving linear systems. However as the size of the problems increase, the complexity of solving them also rises. Either the memory requirement becomes humongous or the execution time is high. The use of multiple machines can contribute better solutions by offering more memory, and reducing the execution time even for very large systems. Multiple machines can be parallel computers or cluster of systems. Parallel computers are a set of processors that work collectively to solve a computational problem. These have thousands of processors, networks of workstations and embedded systems. Parallel computers have very high cost and therefore may not be affordable by scientists and researchers. Machine cluster, on the other hand, are simply multiple machines, used in combination. These clusters are located geographically distant from each other and are referred to as grids. They are heterogeneous systems as these grid systems have different architecture, network parameters, such as bandwidth and latency. A grid may be defined as a set of interconnected local clusters. Each node of the grid performs a separate task. Using these clusters as resources in collection to do the computation for large linear systems is referred to as a grid computing.

The performance of traditional numerical methods, including those which are designed to run on parallel homogeneous machines may be unacceptable as these grids are heterogeneous in nature and therefore become unsuitable for this computing environment. The traditional methods also are not expected to work efficiently on these machines, as there is high dependency during the computation, which requires more communication. These problems catalyzed techniques of solving that work well in a grid computing environment. Such methods are called multigrid methods. This scheme solves differential equations using a hierarchy of discretization. There are two kinds of grids, coarse and fine. The kind of grid used to solve the solution is related to the error introduced in the system. A component that appears smooth on a fine grid may appear not-so-smooth on a coarse grid. Therefore a good strategy is to use multigrids to improve the convergence rate of iterative methods. In addition, they are also more scalable as they are optimal in terms of the number of operations required, i.e. $O(n)$. There are two kinds of multigrid methods; the first type is called the Geometry multigrid (GMG) which uses two or more meshes. The second type is called the Algebraic multigrid (AMG) which doesn't necessarily have to use multiple meshes necessarily; rather, it mimics what happens on a coarse mesh. These kinds of multigrid methods are used for linear systems. For non-linear systems, a multigrid scheme is called Full Approximation Scheme (FAS). For the scope of this work, we will focus on linear system schemes. The main advantage of using multigrid scheme is that the accuracy obtained is the same as fine mesh, and the convergence

is in between a fine mesh and a coarse mesh for two-grid multigrid method. multigrid methods are among the most powerful solving methods for sparse linear systems.

multigrid schemes use a sequence of coarse grids to accelerate the fine grid solution. The ultimate solution must be obtained on the finest mesh. The coarse mesh is used only to accelerate the convergence process and not to obtain the ultimate solution. For simplicity, let us consider a 2-grid mesh. It has two mesh levels, one with the fine mesh and the other with the coarse mesh. The first step is to set up these meshes. Next is to solve the equations on the fine mesh to partial convergence. The solving has to be done for partial convergence, or else we will end up solving the system at the same convergence speed, with no improvement. Solving to partial convergence will propagate some error, which would be taken care of in the future steps of this process. The way partial convergence is executed varies. One common technique is to stop when the residual drops by a factor of two, meaning the ratio of residual obtained by the initial residual is less than 0.5. The next step involves calculating the residual for the fine mesh. Once it has been calculated, the residual is transferred to the coarse mesh. In the next stage, solve to partial convergence is performed. Next the correction over the coarse mesh points is computed. At this stage, a solver that can reduce the error to zero is preferred. And finally, the algorithm transfers the correction from the coarse mesh to the fine mesh. So, the main role of the coarse grid is to compute an improved initial guess for the fine-grid relaxation.

5.1.1 Finite Element Tearing and Interconnecting and its parallel solution algorithm (FETI)

Direct solvers are more suitable for smaller problems and iterative solvers are preferred for larger problems. This solver uses a hybrid approach: It uses an iterative solver and then breaks the problem into sub-problems and applies direct solvers on those sub-problems. This algorithm [35, 65] uses a domain decomposition approach for solving the given linear system for finite element solution in a parallel fashion. The main problem domain is partitioned into non-overlapping sub-domains. These sub-domains are fully independent, which makes FETI suitable for parallel computing. Each one of these sub-domains is assigned to a separate processor. These sub-domains are connected later on by using Lagrange multipliers on neighboring sub-domains. Applying a direct solver to solve the unknowns present in that domain solves each of the sub-domains. The solution of the sub-domain problems is then parallelized. This improves the chances of convergence for a given overall. Figure 10. Stage (a) shows the decomposition into four sub-domains, for instance. Stage (b) shows the splitting of Lagrange multipliers and forming clusters.

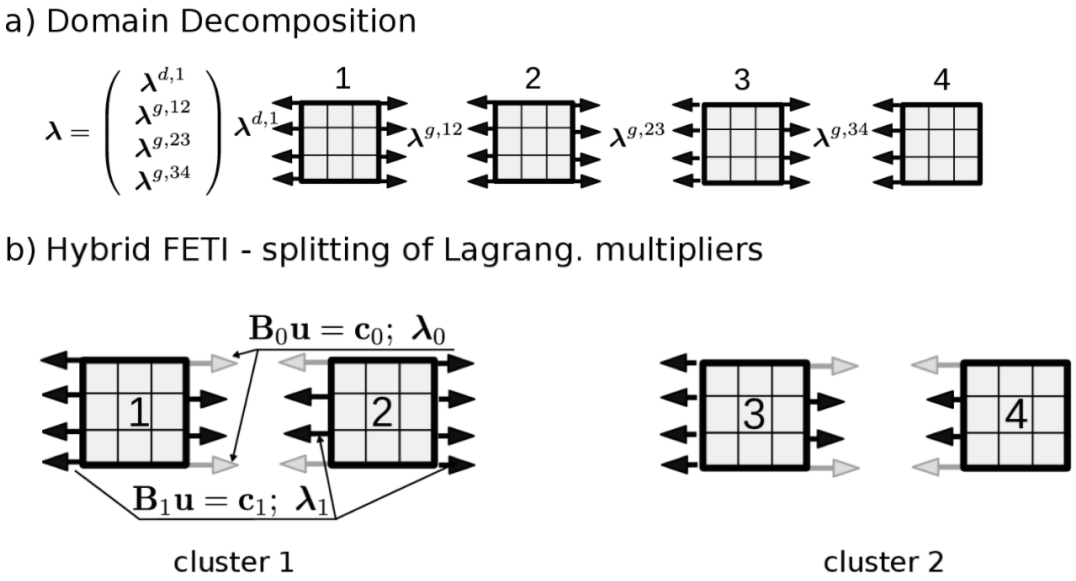

Figure 10: Domain Decomposition in FETI.

This also means that if the given linear system can be solved by a direct method, FETI would be slower than that; however in comparison to an iterative method, it usually performs much better and has better chance of convergence, as it is using direct solvers on the sub-domain. In addition, it also uses many fewer communication calls between processors as compared to the other direct decomposition methods.

5.2 Hierarchical and Nested Solver Methods

Iterative methods suffer from scalability issues, because irrespective of the solving technique, at least one vector inner product is required in each iteration. Hierarchical methods handle this issue by reducing the number of global inner products and replacing them with local inner products, as they are inexpensive. Hierarchical methods are a generalization of Block Jacobi and Additive Schwarz Methods. Applying Krylov methods on smaller blocks solves each block. Each level in the hierarchy applies the same solver method for all sub-problems. They are hierarchical in the sense that this block-subblocks hierarchy can continue to form smaller and further smaller blocks.
Nested Krylov methods are a generalization of inner-outer iterative methods. These methods use Krylov method as an inner method and another Krylov method as an outer method to solve the system. They avoid inner vector products completely by using inner iterations that do not require performing global and local inner products at all.

In [53] the authors apply FGMRES method for the hierarchical strategy, with 2 levels of hierarchy and FGMRES, BiCGStab and Chebyshev as the nested Krylov methods. The way global inner vector products are reduced is by applying a hierarchical approach or nested multi-layer iterations. This can reduce the overall solve time and improve scalability for large-scale applications. The minimum dimension that can be used with these approaches are $O(10^6)$ using at least $O(10^4)$

cores.

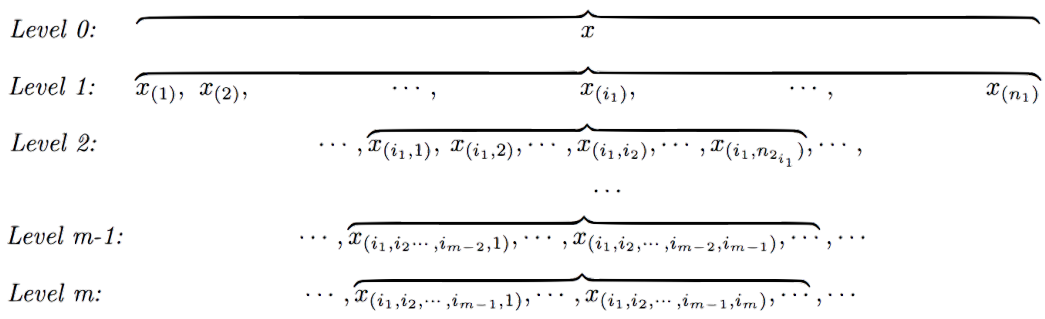

Figure 11: Hierarchical vector partition

In the hierarchical approach, the initial global vector x is partitioned into a hierarchy and also the total cores are partitioned with the same hierarchy. A segment at an upper level is the parent of the segments at lower levels. The upper level segment is inherited by the lower level segments. Figure 11 shows the hierarchical vector partition for vector x. At each level m, the partitioned segment is given as: $x_{(i_1,\ldots,i_{m-1},i_m)}$. In this the first $m-1$ indices, i.e. $(i_1,\ldots,i_{m-1)}$ are inherited from the parent and the rest i_m are its own index. Until convergence on all cores is achieved, it applies Krylov solvers at each level. The lower level solvers are responsible for smaller subsets of the problem sub-domain forming local sub-systems. The top-level solver iterates over all variables of a global problem. The iterations proceed starting from the top level to the lowest level, by doing an inexact solver for each local sub-system. Inner solvers can be chosen based on the characteristics of the sub-domain problem that improves the overall performance of the systems. This gives the flexibility of using the solvers, which are most suitable at that level of the hierarchy.

Figure 12 shows the partitioning for a two level hierarchy. For simplicity a a global dimension of 64 is shown across four cores. As shown in the figure, it divides the problem such that the lower levels are in charge of solving smaller sub-problems.

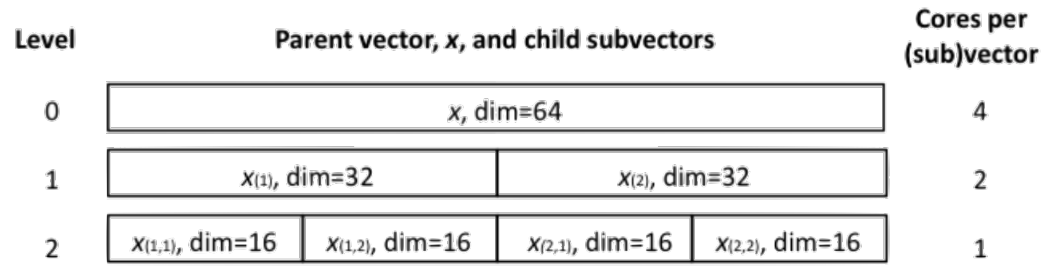

Figure 12: Hierarchical vector partition with two levels

Layered nested Krylov methods, also called inner-outer iterative methods, use a different iterative method as a preconditioner for each iteration in one of the following two ways:
First, a flexible Krylov method is used as an outer iterative method such that it allows variable preconditioning. The inner iterative method, in this case can be any iterative method, like GMRES and Conjugate Gradient. Second, the outer iterative method can be any Krylov method and the inner iterative method is a fixed preconditioner. A fixed preconditioner is one which is applicable only if the number of linear operations to be performed are fixed. A variable preconditioner is one, which can be employed if the number of these operations is not fixed. In [53] BiCGStab is used with Block Jacobi preconditioner, using one block per core and ILU(0) as the local solver. They consider BiCGStab, GMRES and Chebyshev solvers. BiCGStab was preferred over GMRES because the former avoids repeated orthogonalizations that have high communication cost associated with them. They used Chebyshev successfully with other Krylov methods as it requires no inner products. These methods reduced overall simulation time drastically, due to the combined effect of reduced global synchronization, due to few global inner products and stronger inner hierarchical/ nested preconditioners.

6 Multi-method Solver Systems

In a single solver scheme, the choice of the solver method is made by experts or dependant on the resources available online and documentation provided with different methods. But in many cases, there is often no single solver that is consistently better, even for problems from a single application domain. There is also no guarantee that the solver method applied will converge. These challenges generated the idea of a solving strategy that involved more than one solver algorithm. The earliest work [62, 63], which suggested that the efficiency of a system is expected to improve with polyalgorithm solvers, used three basic solvers. [63] provides the design details of a polyalgorithm for automated solution of the equation $F(x) = 0$. The challenges for the work include the small number of problems available for solving. In addition, the system did not find all the roots for the equation mentioned above and for some of the roots it found out, they were questionable.

In this section we talk about different types of multi-method solver approaches, namely composite solvers, poly-iterative solvers and adaptive solvers. Multi-method linear systems include a variety of techniques, such as composite solvers, iterative solvers and adaptive solvers. Below are the papers that cover the variety of multi-method solver techniques that are popular for use. Figure 13 shows the comparison of various solve schemes in terms of time and number of solvers for single-method solvers and multi-method solvers.

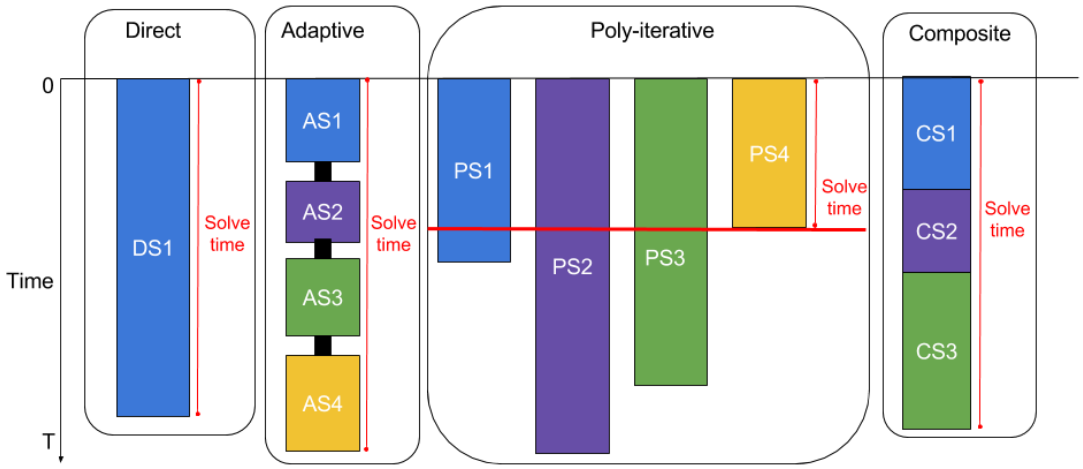

Figure 13: Comparison of various solve schemes

6.0.1 Composite Solvers

A composite solver approach is the one in which basic solver methods are sequenced in an ordered fashion [16, 15, 17, 18, 2]. The first choice for solving the system is the first solver method in the sequence. If a method fails, then the next method in the sequence is invoked. This continues to happen until the problem is solved successfully. The composite algorithms are developed to have efficient and robust solvers as composites of multiple solver methods. [16] achieves this by using multiple preconditioned iterative methods in sequence to provide a solution. The solution obtained by this strategy is believed to be reliable and to have a good performance in parallel. This scheme has been tested in a driven cavity flow application for performance. As mentioned in [2, 16], the reliability of a method is given as $r_i = 1$ - f_i, where f_i refers to the failure rate. The running time of the composite scheme depends on the sequence of the solver methods. The worst-case time scenario for the composite solver scheme is when it needs to attempt solving the system with all the solver techniques in the given sequence. This can be given by:

$T_\pi = t_{\pi(1)} + f_{\pi(1)}.t_{\pi(2)} + f_{\pi(2)}.t_{\pi(3)} + ...(f_{\pi(1)}...f_{\pi(n-1)})t_{\pi(n)}$.

To have minimum worst-case running time among all the possible combinations possible, the base methods are arranged in the sequence in the increasing order of their utility ratio, u_i which is given by the ratio t_i/r_i. For computing this ratio, r_i is substituted as shown below and using estimates of t_i with some sampling technique:

$r_i = 1 - f_i$

This also means that this technique uses knowledge obtained in the past, which enables using domain-specific knowledge for the selection of solvers. This system maintains the past performance history and allows monitoring of system performance.

The solvers are then arranged in the increasing order of their utility ratio, u_i. They use a simple sampling technique for the optimal composite by computing this ratio from running all their solver methods in the sequence on a small dataset and obtaining the mean of the time taken per iteration by the solvers and the failure rates.

The software architecture that supports this strategy is modeled in Figure 14. It has the following components: solver proxy, non-linear solvers, linear solvers, ordering agent and application driver. The proxy linear solver method acts as an intermediate between the non-linear solver algorithm and linear algorithm. The proxy, linear solvers and non-linear solvers have the same solver interface to make it easy to use multiple solvers. This proxy interacts with the ordering agent to choose the linear solvers based on the ordering strategy. The proxy is used with Newton-Krylov solver. They use the following four set of base solution methods: (1) GMRES(30), restricted additive Schwarz method (RASM)[1] with Jacobi sub-domain solver (2) GMRES(30), RASM[1], with SOR sub-domain solver (3) TFQMR, RASM[3] with no-fill ILU sub-domain solver and (4) TFQMR, RASM[4] with no-fill ILU subdomain solver. The numbers in brackets denote the degree of overlap.

[15]

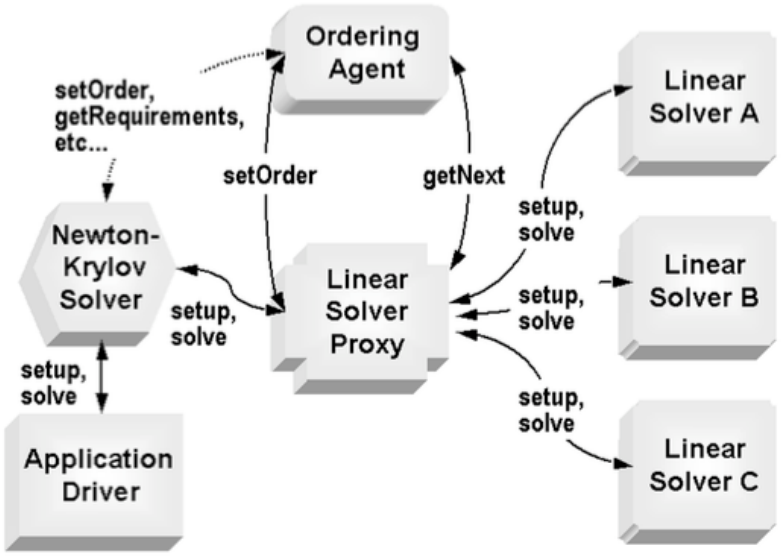

Figure 14: multi-method software architecture.

6.0.2 Adaptive Solvers

In this approach [52, 26, 27, 14, 33] only one solver is used, and it is selected as the most suitable solver dynamically, based on the match of the solver with the characteristics of the linear system under consideration. This technique adapts the solver method during a simulation, based on the changing attributes of the problem. The difference this approach has over the composite solve approach is that it uses only one base solver for each linear system.

Numerical properties for a system change at each iteration of a given problem and so does the choice of the solver and the selection criteria. Linear solvers [52] are selected at each iteration based on the characteristics of the problem emerging at each level; given the nature of the problem

at that stage, the decision of the best suited solver is taken at each stage. This strategy applies a different preconditioner during different simulation stages, maintaining a low overall time for finding the solution. This is done by the combination criteria for different methods. For instance, if there is a more robust method at one stage, then the other stages had methods that were faster to compensate the time taken by the first method. They used GMRES(10) with a point-block ILU(1) preconditioner. In this approach, the chances of obtaining a solution are increased by the use of robust methods, and the total time for the solution is acceptable. [14] is an extension of the previous work to solve a more complex parallel application to show that the adaptive poly-algorithmic approach is parallelizable and scalable. They used four linear solvers:

(1) GMRES with a Block Jacobi preconditioner and SOR as a subdomain solver, called GMRES-SOR.

(2) Bi-conjugate gradient squared (BCGS) with a BJ preconditioner and no-fill incomplete factorization (ILU (0)) as a sub-domain solver, called BCGS-ILU0.

(3) flexible GMRES (FGMRES) with a BJ preconditioner with ILU (0) as a sub-domain solver, designated as FGMRES-ILU0.

(4) FGMRES with a Block Jacobi preconditioner that uses ILU (1) as a sub-domain solver, called FGMRES-ILU1.

The switch is made on the following two indicators: 1. The non-linear residual norm is calculated and assigned to the following four categories: (a) $||f(u)|| \geq 10^2$, (b) $10^4 \leq ||f(u)||<10^2$, (c) $10^{10} \leq ||f(u)||<10^4$, and (d) $||f(u)||<1010$ A change in the solver is made when the simulation moves from one category to another and the solver method is moved up or down accordingly. 2. Average time per non-linear iteration: The base solver methods are arranged in increasing order of their corresponding average time per nonlinear iteration.

[26] has a different approach. It uses statistical data modeling to make the solver choice automatically. It combines different solver techniques with different preconditioners and different parameters as well.

6.0.3 Poly-iterative Solvers

Poly-iterative solver approach uses multiple solvers applied simultaneously to the system so that the chances of getting a solution increase. If one solver fails, one of the other solvers from the system may provide a solution.

One of the earliest suggestions made in [62] for poly-iterative solver strategy was made in the late 1960s. This strategy is based on selecting the solver based on the problem size and the user's specifications about the problem and the accuracy level expected from the system. If the user specifies no information, then the size is used to pick the solver. If it is a small matrix, with less than 15 rows and 15 columns, the solver chosen is LU decomposition. If this method fails, then the solution obtained just before failure is used as the initial guess for SOR. If this method also fails, the user is provided with the summary and prompted for further instructions to solve. The user may lower the accuracy level to accept somewhat less acceptable solution or allow longer computations for solving the system.

For problems of larger size (considered large in that decade), more than 80 rows and columns, SOR is applied. If this fails, SOR is applied again, but this time on the product of matrix transpose and the original matrix. If this strategy fails then the user is asked for further instructions similar to the small matrix scheme.

For problems of intermediate size, the properties of bandedness and diagonal nature are inves-

tigated. If either of these properties are valid for the problem in consideration, SOR is applied. If that fails, LU decomposition is used. If that fails as well, then the system relies on user feedback for accepting lower accuracy or allowing longer computations.

The work in [9] with the poly-iterative approach [62, 64, 34] mentions the advantages of using a poly-iterative approach in parallel. First being an increased probability of finding a solution. The second being increased performance resulting from an efficient matrix-vector product. In addition, once any one of the solver methods has converged, the process can be terminated. This algorithm uses three solver techniques: 1. QMR 2. CGS 3. BiCGSTAB. These methods start computing the inner product, and then perform the vector updates and finally a preconditioner solve. All these methods are applied simultaneously and as soon as one of them converges, the iteration is stopped for all other methods. The cost per iteration is the sum of the cost of the three methods. In case if a method fails, it is removed from the iterative scheme. This strategy takes more time than the best method, but is preferred, sometimes it has a higher probability of finding the solution. The extra time is due to the communication cost involved in this strategy, although making communications global may reduce it. Another situation in which this method incurs higher cost is when one of the methods is comparatively more expensive than the others and it is not the first method to converge nor it fails. However, this strategy is more beneficial in a parallel implementation, as this approach aligns the mathematical operations of the solver methods and combines the communication stages to make it more efficient. Figure 15 shows the sequence of operations and how the communication is combined.

The work in [64] provides methods that can be used for automatic solution of $F(x) = 0$ where $F(x)$ is a non-linear equation for one variable. This is one of the oldest solver methods, done in 1968, when more than the size of the problem, solving a problem was the focus. It uses three base methods: Secant method, half-interval method and descent method.

1. Secant method: Secant solver is one of the oldest solver methods in the history of numerical analysis. It is a root-finding algorithm that finds roots in succession to get an approximation of the root. This method is an approximation of the Newton's method. The secant method is as follows:
   $x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1}-x_{n-2}}{f(x_{n-1})-f(x_{n-2})} = \frac{x_{n-2}f(x_{n-1})-x_{n-1}f(x_{n-2})}{f(x_{n-1})-f(x_{n-2})}$

2. Half-interval method: This method works in conjunction with the previous method. It applies secant method for a short run for both the intervals; each interval being half of the total time. If none of these secant method work, the point at which the sign changes is classified as a discontinuity.

3. Descent method: This is descent on the absolute function of the non-linear equation used to alter the location of the root. It is used only when a root is found by the secant method and cannot be used standalone.

7 Software Examples

In this section I describe some of the software packages that are commonly used for solving sparse linear systems. While there are significant overlaps in functionality in some cases (e.g., preconditioned Krylov methods in PETSc and Trilinos), the packages discussed here cover different sets of algorithms and incorporate different implementation strategies.

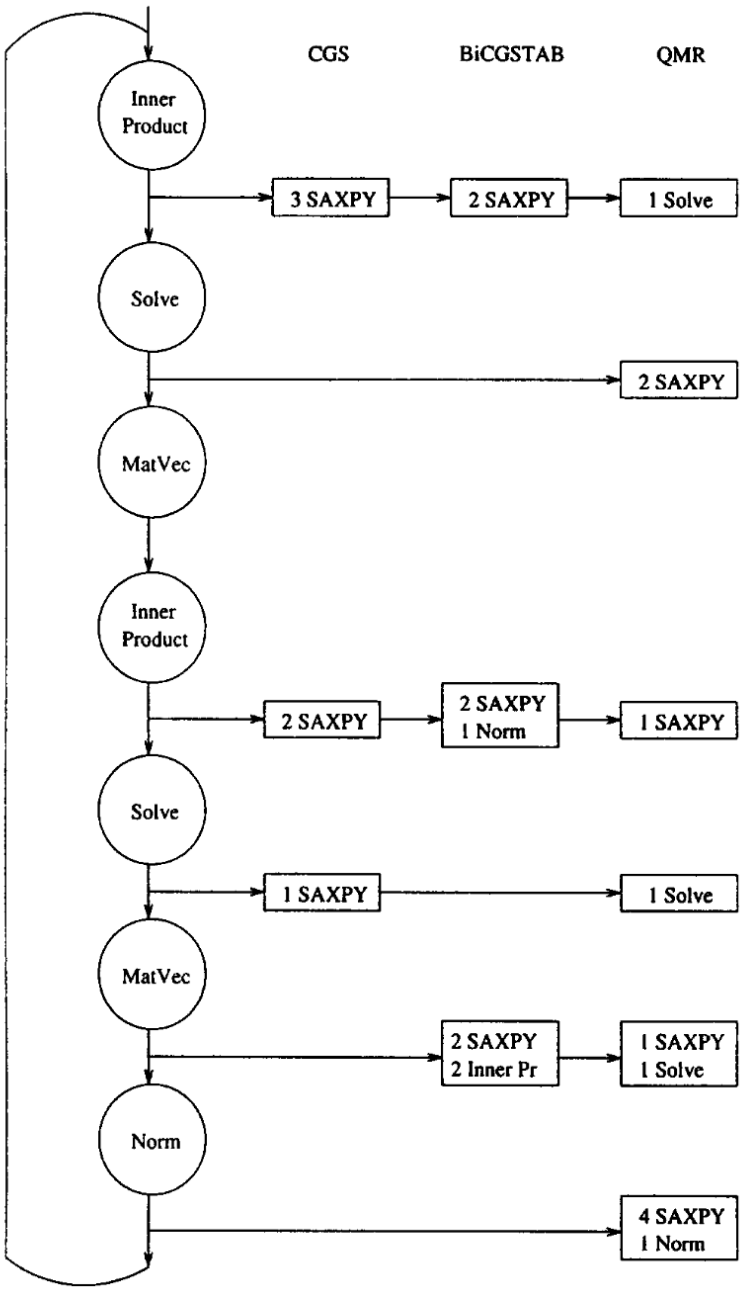

Figure 15: Sequence of operations.

7.1 PETSc

Portable, Extensible Toolkit for Scientific Computation (PETSc) [8] is a popular toolkit for linear system solvers and preconditioners. It can run on different kinds of architecture and different operating systems. It is portable to any parallel system that supports MPI. It provides scalable solutions and is efficient and has unique features which give the library widespread popularity. It provides a wide variety of scalable parallel preconditioners, krylov subspace methods. Table 16 shows the list of solvers and preconditioners offered by PETSc.

Capability	Algorithm
Preconditioners	Jacobi point block Jacobi block Jacobi additive Schwarz
Incomplete factorizations	ILU dt
Matrix-free	infrastructure
Multigrid	infrastructure geometric (DMDA for structured grid) geometric/algebraic structured geometric classical algebraic (BoomerAMG/hypre) classical algebraic (ML/Trilinos) unstructured geometric and smoothed aggregation
Physics-based splitting	relaxation and Schur-complement least squares commutator
Approximate inverses	approximate inverses
Substructuring	balancing Neumann-Neumann BDDC
Krylov methods	Richardson, Chebyshev, conjugate gradients, GMRES, Bi-CG-stab, transpose free QMR, conjugate residuals, conjugate gradient squared, bi-conjugate gradient , MINRES, flexible GMRES, LSQR, SYMMLQ, LGMRES, GCR , Conjugate gradient on the normal equations

Figure 16: Solvers and preconditioners offered by PETSc.

PETSc also has profiling abilities. PETSc gives its users the flexibility to change the level of abstraction to what suits their problem the best. It has a plug-in architecture to offer extensibility to its users. This means additional implementations can be added and made available for other users. Figure 17 shows the hierarchical organization of the toolkit. Starting from the bottom level, the lowest level has the standard library for PETSc, such as MPI for communication, BLAS, LAPACK for linear algebra computation. The next level in the hierarchy is the lightweight-profiling interface, which gives information such as where the code is spending its time during solution, which is useful to application programmers, and indicates how their program is organized. The next level has the basic building blocks, i.e, the data objects that PETSc uses, such as matrices, vectors, and indices to access blocks. The DM module at this level is how PETSc coordinates interaction between what happens with meshes and discretization in differential equations. It is an adapter class that communicates information from discretization in the mesh to the solvers. It makes using the solvers easier for the toolkit users. The next level has the linear solvers, which are the Krylov subspace

methods and preconditioners. PETSc also offers non-linear solvers, which form the next level in the hierarchy. Optimization is provided by the TAU [71] component of PETSc for linear solvers within PETSc algorithm and basic building blocks. The ODE intergrators interface non-linear solvers and sometimes-linear solvers directly, depending on the problem.

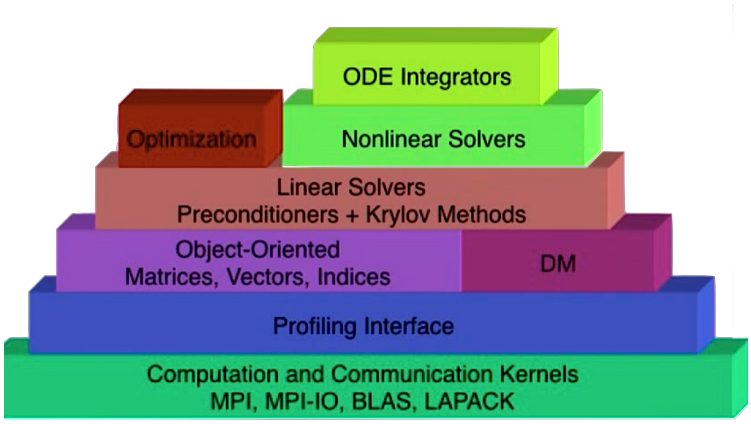

Figure 17: PETSc Components.

7.2 Trilinos

Trilinos [43] is an object-oriented software framework for solving science and engineering problems. It runs on laptops, workstations, and small and large clusters and supports MPI. It offers a variety of linear and non-linear solvers and preconditioners. Figure 3 shows the list of solvers and preconditioners that are offered.

Solver and preconditioner interfaces	**Names of solvers and preconditioners**
Iterative linear solvers	AztecOO, Belos, Komplex
Direct sparse linear solvers	Amesos, Amesos2, ShyLU
Direct dense linear solvers	Epetra, Teuchos, Pliris
Iterative eigenvalue solvers	Anasazi, Rbgen
ILU-type preconditioners	AztecOO, IFPACK, Ifpack2, ShyLU
Multilevel preconditioners	ML, CLAPS, Muelu
Block preconditioners	Meros, Teko
Nonlinear system solvers	NOX, LOCA, Piro
Optimization (SAND)	MOOCHO, Aristos, TriKota, Globipack, Optipack
Stochastic PDEs	Stokhos

Table 3: Trilinos Solvers and Preconditioners.

The Trilinos solvers interfaces are described briefly as follows. The AztecOO interface includes the iterative linear solvers namely, Conjugate Gradient, GMRES, BiCGStab and incomplete factorization preconditioners. AztecOO is the improvement over Aztec offered by Trilinos. Belos is

an interface for a variety of solvers, such as GMRES, Conjugate Gradient, block and pseudoblock solvers, hybrid GMRES. Belos and AztecOO have some overlapping solvers, however Belos is believed to be better than AztecOO interface. Amesos2 is a direct solver interface for Tpetra. This includes SuperLu solver and all its variants. ShyLU is a scalable LU which is a hybrid of direct and iterative solver for robustness. Ifpack(2) includes Incomplete factorization, Chebyshev, Block relaxation, domain decomposition preconditioners.It uses Epetra and Tpetra for matrix and vector calculations. Ifpack(2) is a Tpetra version of Ifpack. ML includes the multi-level preconditioners which are multigrid and domain decomposition. These are the ones that are considered for scalable performance of applications. MueLu is similar to ML but better in the sense that it can work with Epetra and Tpetra objects as well.

7.3 MUMPS

MUMPS is a multi-frontal massively parallel sparse direct solver library that uses a multi-frontal technique, shown in Figure 18. It focuses on providing solutions for symmetric positive definite matrices and symmetric and non-symmetric matrices. MUMPS solves large systems of linear equations of the form $Ax = b$ by factorizing A into $A = LU$ for non-symmetric matrices and $A = LDL^T$ for symmetric matrices. Here L and U are the lower and upper triangles of A respectively and D is the diagonal matrix. It is a three-phase approach. The first step is the analysis phase in which it analyzes the matrix structure and prepares for factorization. Then it preprocesses the matrix and performs the data factorization. The second phase is the phase of factorization, where it factorizes the matrix A based on the properties of the matrix. A non-symmetric matrix A is factorized into $A = LU$. If it is a symmetric matrix and it is also positive definite, the factorization is as follows: $A = LL^T$. If it is symmetric indefinite then factorization changes to $A = LDL^T$. And the final phase is the solution phase, in which it performs the forward and backward substitutions $Ly = b$ and $Ux = y$.

7.4 Self-Adapting Large-Scale Solver Architecture (SALSA) Solvers

SALSA [28] is a self-adapting solver technique which has several levels on which the computational choices for the application scientist are automated. The choice of solver technique can be made based on the nature of data and on the efficiency of the available kernels on the architecture under consideration to facilitate tuned high-performance kernels. One of the advantages of this scheme is that it is expected to increase its intelligence overtime. It is self-adapting software that remembers the results of the runs and learns over time. There are three levels of adaptivity, as follows.

1. Kernel level: It can be done in one-time installation and is independent of the data given by the user.
2. Network level: Some level of interaction with user data.
3. Algorithm level: At this level, analysis is done dynamically based on the user data.

SALSA is a component-based framework; Figure 19 shows the four components of the framework. An Intelligent Agent that includes an automated data analyzer to reveal necessary information about the structure of the data, a data model for expressing this information, and a self-adapting decision engine that can combine the meta data to choose the best library and algorithm for solving.

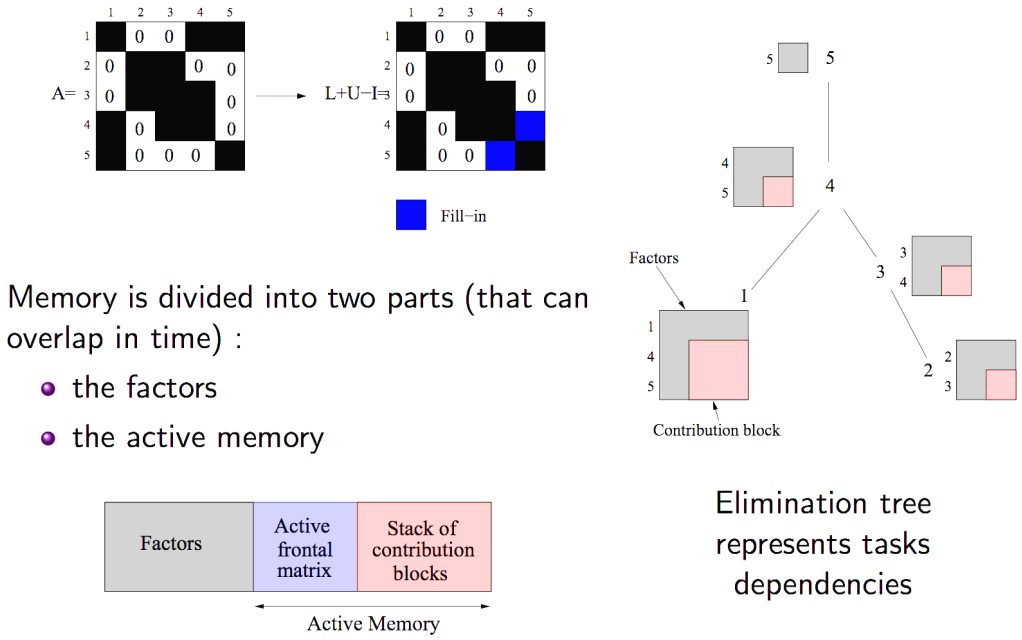

Figure 18: Multi-frontal solving technique.

The analysis module computes the properties of the input. The history database records all the information related to the intelligent component along with the data that each interaction with a numerical routine produces like the algorithm. Metadata vocabulary provides information about the data and the performance profiles. It is used to build the history database and supports building an intelligent system.

7.5 Linear System Analyzer Solvers

The Linear System Analyzer (LSA) [20] is a component-based problem-solving environment for large sparse linear systems. The components LSA provides are broadly categorized in four categories, IO, Filter, Solver and Information. IO is for feeding the problem into the system and getting the solution out of the system. The user also feeds various parameters and settings for solving such as, the relaxation for solving, what solver to be used for solving, etc. Although this system takes a lot of input from the user apart from the problem to be fed as input, it provides settings to choose default parameters for the various solving techniques and other settings shown on the interface. Figure 20 shows a sample LSA session. Filter is for providing filtering of data or system manipulation in other words, such as scaling, eliminating entries based on the size, etc. Solver is for actually getting the system solved. LSA offers four choices to the users to use for solving. They are as follows:

1. Banded: A matrix has a banded structure if its rows and columns can be permuted such that the non-zero entries form a diagonal band, like exhibiting a staircase pattern of overlapping

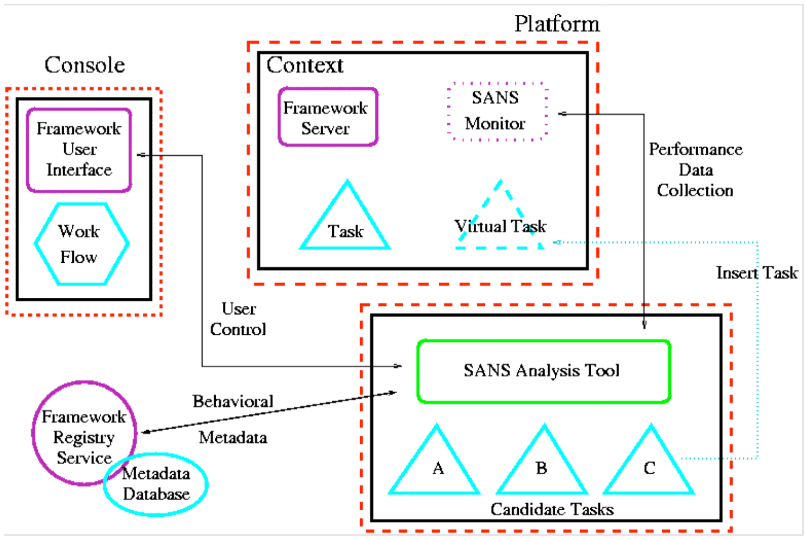

Figure 19: Component based framework.

rows. It converts the system to banded structure and solves the system, with the new data structure and uses LINPACK [29] routines for solving. LINPACK, LINear algebra PACKage is a Fortran package developed in 1970's. It uses BLAS for the underlying routines.

2. Dense: It converts the system into a dense 2D array data structure and then solves it using Lapack [3] routines. system to a dense 2D array data structure.

3. SuperLU: SuperLU [50] is a solver library for getting direct solutions for large, sparse, non-symmetric systems of linear equations.

4. SPLIB: They use preconditioned iterative solvers offered by SPLIB library [21]. This library had 13 solvers and 7 preconditioners when this research was performed.

Figure 21 shows the LSA architecture with its four components namely, user control, manager, communication subsystem, and information subsystem. This approach provides parallelism between components, which supports solving large problems by simultaneously using the computational resources of multiple machines. This system allows comparisons of different solver methods and support to facilitate practical solution strategies.

The user control module is a Java interface. A new system is fed as input in the first component in the module "NewSystem". Meanwhile, the matrix is scaled and reordered, simultaneously on different machines. These actions are performed through the user control module, which also has options for choosing parameters for all the modules in the interface. The scaled problem is sent to SuperLU and the reordered version is sent to SPLIB where SuperLU and SPLIB are the two component subinterfaces. LSA has an option of running multiple solvers on a single system in order to compare these techniques and use them for research purposes.

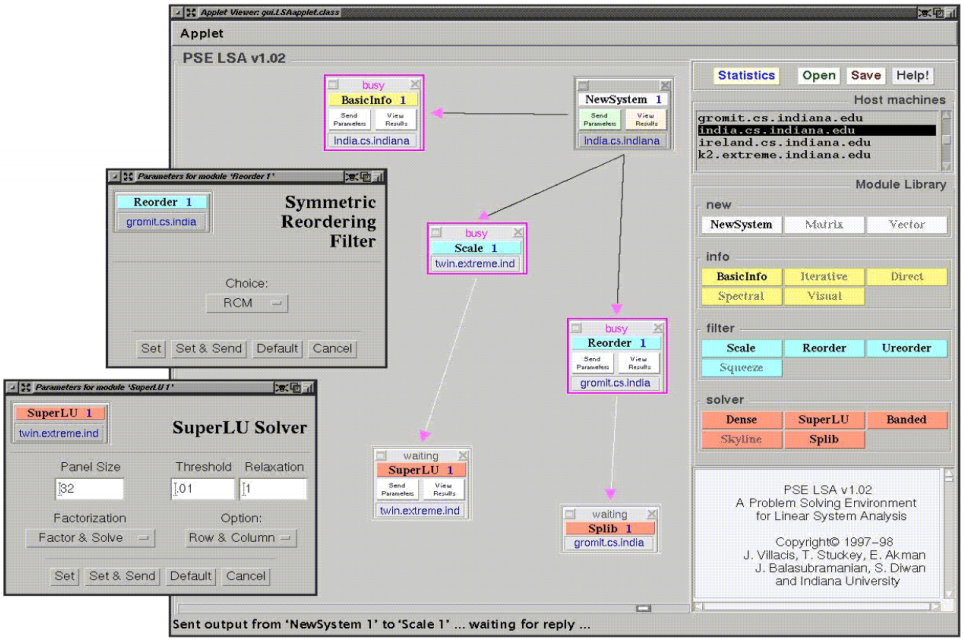

Figure 20: Sample LSA Session.

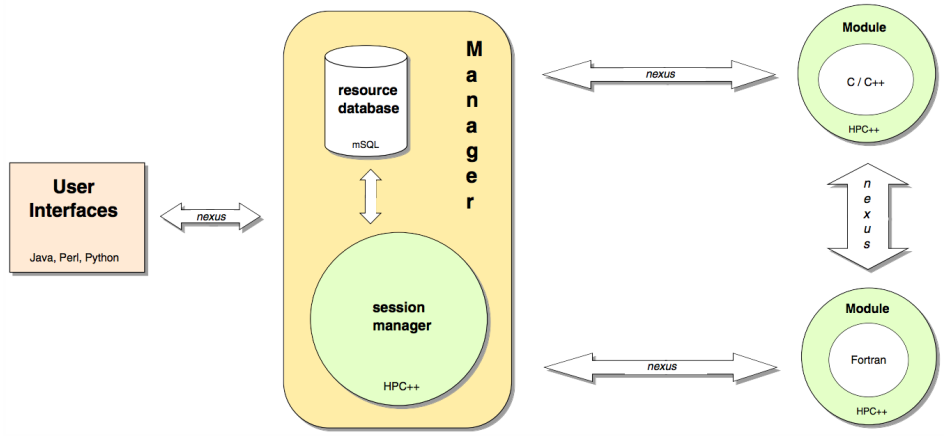

Figure 21: LSA Architecture.

The LSA manager collaborates control and resource management. It establishes a component network to facilitate multiple user control systems with a single LSA session. It also assigns unique identifiers and maintains the database of various machines and components.

The next module is the communication subsystem, Nexus [36] which is a cross-platform system for facilitating parallel applications and distributed computing. LSA uses a many libraries for solvers and preconditioners that are written in different programming languages and therefore there needs to be a module that handles this and make it robust as a mixed language system. Nexus provides this bridge between the different languages used in LSA.

The Information subsystem module provides any information that the user may want about the solving process except the undesirable information. The results are shown in the form of a summary with the performance metrics for that scenario. There is a small description provided, along with the details whether the event was successful or failure or there was a warning. The user is also redirected to more information, in case if she wants more details.

8 Performance Modeling Approaches for Solvers

As illustrated in the previous section, there are various options available for solving a linear system. The choice of selecting the solvers is made on past research and expertise. There are some common sets of metrics that are used to measure the performance of a model which are listed as follows.

1. **Ranking**: The first set of metric is based on ranking. The model is rewarded for generating predicted values with the same pairwise ordering as the actual pairwise ordering of the values in consideration. Area under the curve is one such metric. Ranking techniques are good for direct solvers, as direct solvers have a unique solution which can be compared with other solutions, if they exist. However, a ranking model would not necessarily be useful for the solver selection work, as the solver methods are iterative in nature which makes them approximations of the solution. Therefore, ranking may not be possible, as ranking approaches tend to be very solver specific. For instance, with iterative solutions there can be multiple correct solutions and more than one method can be equally good.

2. **Regression**: The second set of metrics is regression-based, in which, if the predicted value is close to the actual value, the model is rewarded. If the predicted value is far away from the actual value, the model is penalized. An example of such a metric is Mean Squared Error. The goal of regression is to learn a model that can predict a real-valued target from a vector of features using a prediction function, unlike ranking models, which are classification models that predict the class from a possible set of classes. Regression models are most useful when the actual value of the prediction is very important. One example is a model for generating the estimated price of airplane tickets.

3. **Regression and Ranking combined**: A model that performs well on both these metrics is hard to achieve. In fact, a model with great performance on one metric can perform very poorly on the other metric. One such example would be a case of binary class distribution with one of the labels occurring very rarely; the model can achieve almost perfect regression by always predicting the class that is more frequent. But ranking, on the other hand, is not valuable. Many situations demand good performance in both these metrics or at least acceptable performance for both metrics in the majority of cases. In the past few years, there has been some work done for achieving good performance for both these metrics. This has been

achieved by combining both the above-mentioned approaches in [70]. Combining ranking with regression adds additional useful constraints to the model, which gives an improvement over the regression-only models. Therefore this approach enables improved regression performance, especially in the cases of rare events, as compared to the performance of regression-only methods. With the regression loss $L(w, D)$ and pairwise ranking loss $L(w, P)$, the combined ranking and regression approach optimization problem is given as follows:

$$\min_{w \in R^m} \alpha L w, D) + (1 - \alpha) L(w, P) + \frac{\lambda}{2} ||w||_2^2$$

where D is the dataset, P is the set of candidate pairs, L is the regression loss, w is the weight vector, λ is the parameter that controls regularization and α is the parameter for combined regression and ranking. In the above equation $\alpha \in [0, 1]$ Setting $\alpha = 0$ makes it pairwise ranking-only and setting $\alpha = 1$ makes it regression-only approach. For the optimization to consider both the techniques, α is set to a value in between 0 and 1.

4. **Classification**: Another approach [12] for modeling solver performance is by using machine-learning methods to classify the solvers for a linear system. These methods apply supervised learning on the data and learn about the linear system based on the characteristics of the problem. Once the various machine-learning algorithms are applied, there can be a comparison of the accuracy of the solver prediction and time it took to build the model for each machine learning algorithm. These learning algorithms can then be used to make predictions for which solvers should be used for a new incoming system. In this approach of solver selection, each linear system can be solved with multiple solvers. Based on the time it takes for each solver to solve the system and whether it converged or not the solvers can be classified. The class label can be chosen based on the solve time and a threshold chosen as the acceptable rate for a solver to be considered "good".
   Once a label is assigned for each data point, supervised learning can be performed on the data. To do this, some machine-learning classification algorithms can be used to make the solver selection for the linear problems. Below are some of the methods that that can be used: BayesNet [19], k-nearest neighbor [24], Alternate Decision Trees [13], Random Forests [22], J48 [61], Support Vector Machines (SVM) [23], Decision Stump [47], LADtree [45].

There are other approaches to doing the classification, which include more than two-class labeling. For instance, a strategy with tertiary labeling. However a two class labeling is better than two class labeling for solver predictions for two reasons: given the nature of our system, what seems to be most interesting is whether a solver was good or bad for a given system. A solver being mediocre good for a case is not likely to be of interest to the systems users. Second, classification with three classes of solvers will reduce the number of data points in each category. This may result in the case where there are not enough data points for each category because the categorization is done based on the time it takes to solve a system and it may be the case that most of the solver times don't fall in one of the categories, which will drop the performance.
However, techniques like the combined ranking-regression mentioned above, can be applied to achieve ranking among the solvers along with regression and the 'good' solvers can be ranked based on how much time they took to solve the system. To ensure the suggestions made are realistic, one can verify the ranking results the model produces, with the ranking that one would expect based on the theoretical knowledge we have about the various solver methods.

9 Conclusion and Future Work

In this work I presented the two categories of solvers, direct and iterative that can be used for solving large sparse linear systems. This work briefly portrays the role of preconditioners. The report describes in detail some of the most popular direct and iterative solvers, followed by comparing the two approaches. The report further outlines the time and space complexity and how scalability of these methods introduces a new technique of solving linear systems. Later in the report we saw some popular software that offer different linear systems solvers for solving large sparse systems. In the later part of the report, I elaborate on the two most popular solving schemes. And we also saw some approaches for modeling solver performance in the last section of this report.

Although traditional ways of solving linear systems emphasize using one solver technique for a given problem, studying the two techniques in detail, we have become inclined to see the advantages of using multi-solver technique for solving. This will not only make the system more efficient, it will give the system the power to alter decisions based on the changing characteristics of the system during the solving process. This will help us build a system that is more efficient, robust, generic and reliable. Direct solvers use a single solver throughout the process. In an adaptive solver scheme, many methods are used but at a time only one solver is applied. But it keeps changing the solver based on switching criteria. It runs one solver and then applies the switching check which involves some calculations like convergence rate and increase in the number of iterations. Once this is done, it decides to use the same solver or switch. In the composite solver scheme, the solvers are sequenced in order and everything is preassembled. If the first solver fails, the system switches to the second solver in the order. In poly-iterative approach multiple solvers are applied simultaneously and whichever converges the fastest, terminates the solving process.

In the future, we would like to use Weka and other existing ML packages to investigate the effectiveness of different ML approaches for our problems. In addition, sampling the solver space intelligently is a task we would like to perform in the future, as it sounds more realistic and doable with bigger datasets. As the dataset grows in size, the number of possible combinations for the solvers and preconditioners and their parameter options also grows drastically. This generates a sample space that is very big in size and becomes hard to handle. This means the number of experiments required to explore the sample space is so huge that it requires almost infinite resources, which is not desirable in any situation. This calls attention to sampling the space intelligently and be able to represent the sample space with less data; yet at the same time, also show a true representation of the space. Any kind of approach that involves randomly picking the data points from the space would not be able to perform the desired task. There has been some work done by researchers in this field as in [51], and this is one of the areas which we have started exploring recently and would like to explore more in the near future.

10 Appendix

1. Diagonal matrix: A matrix in which the non-diagonal elements are 0. For instance, the matrix below is diagonally dominant, as the only non-zero elements are present at the diagonal locations.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & -6 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

2. Diagonally dominant matrix: A matrix is diagonally dominant if for each row, the magnitude of the diagonal entry in that row is larger than or equal to the sum of the magnitudes of all the other non-diagonal entries in that row.

3. Order of a matrix: The number of rows and columns of a matrix are referred to as the order of the matrix. For instance, a matrix with 4 rows and 5 columns has an order of $4x5$.

4. Singular matrix: A matrix whose determinant is zero. For instance, the matrix given below has a determinant 0, which makes it singular matrix.

$$\begin{bmatrix} 2 & 4 \\ 4 & 8 \end{bmatrix}$$

The determinant of the matrix can be given by the Laplace formula:

$$determinant(A) = \sum_{\sigma \in S_n} -1^{N(\sigma)} \prod_{i=1}^{n} a_{i,\sigma_i}.$$
$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma_i}.$$

5. Triangular matrix: A square matrix with the following special characteristics: Lower triangular matrix: The square matrix in which all the elements above the diagonal are zero. Upper triangular matrix: The square matrix in which all the elements below the diagonal are zero.

6. Square matrix: A matrix with the same number of rows and columns.

7. Symmetric matrix: A square matrix that is equal to its transpose, so $A = A^T$. An example of a symmetric matrix is shown below:

$$\begin{bmatrix} 1 & 2 & 4 \\ 2 & -3 & 8 \\ 4 & 8 & -9 \end{bmatrix}$$

8. Factorization: Original matrix is decomposed into multiple smaller matrices. The original matrix can be obtained by the product of the smaller matrices.

9. Ill conditioned matrices: Matrices, which have a very large condition number, are called as Ill conditioned matrices. Matrices with a small condition number are referred to as well-conditioned matrices.

10. Bidiagonalization: The process of converting a matrix into a bidiagonal matrix, which is, a matrix in which the non-zero elements are along the main diagonal of the matrix and either the diagonal below or above the main diagonal. For example, the matrix shown below is bidiagonal.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 \\ 0 & 1 & -6 & 0 \\ 0 & 0 & 3 & 2 \end{bmatrix}$$

11. Tridiagonal matrix: A matrix in which the non-zero elements are along the main diagonal of the matrix and along the diagonal below and above the main diagonal. For example, the matrix shown below:

$$\begin{bmatrix} 1 & 5 & 0 & 0 \\ 2 & 4 & 4 & 0 \\ 0 & 1 & -6 & 1 \\ 0 & 0 & 3 & 2 \end{bmatrix}$$

12. Orthogonalization: The process of finding a set of orthogonal vectors in a given subspace.

13. Symmetric positive definite matrix: A matrix is symmetric positive definite if $A = A^T$, A^{-1} exists, all its Eigenvalues are positive and all elements of A are greater than zero.

References

[1] L. M. Adams. Iterative algorithms for large sparse linear systems on parallel computers. 1982.

[2] E. Alba and J. M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, et al. Lapack users guide. society for industrial and applied mathematics, philadelphia, pa. Technical report, ISBN 0-89871-447-8 (paperback), 1999.

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

[5] O. Axelsson. *Iterative solution methods*. Cambridge university press, 1996.

[6] A. Baker, J. Dennis, and E. R. Jessup. Toward memory-efficient linear solvers. In *International Conference on High Performance Computing for Computational Science*, pages 315–328. Springer, 2002.

[7] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted gmres. *SIAM Journal on Matrix Analysis and Applications*, 26(4):962–984, 2005.

[8] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, et al. Petsc users manual revision 3.5. *Argonne National Laboratory (ANL)*, 2016.

[9] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine. Algorithmic bombardment for the iterative solution of linear systems: a poly-iterative approach. *Journal of Computational and applied Mathematics*, 74(1):91–109, 1996.

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

[11] S. Bhowmick. *Multimethod solvers: algorithms, applications and software*. Pennsylvania State University, 2005.

[12] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of machine learning to the selection of sparse linear solvers. *Int. J. High Perf. Comput. Appl*, 2006.

[13] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of alternating decision trees in selecting sparse linear solvers. 2010.

[14] S. Bhowmick, D. Kaushik, L. McInnes, B. Norris, and P. Raghavan. Parallel adaptive solvers in compressible petsc-fun3d simulations. In *Proceedings of the 17th International Conference on Parallel CFD*. Elsevier, 2005.

[15] S. Bhowmick, L. McInnes, B. Norris, and P. Raghavan. The role of multi-method linear solvers in pde-based simulations. In *Computational Science and Its ApplicationsICCSA 2003*, pages 828–839. Springer, 2003.

[16] S. Bhowmick, L. McInnes, B. Norris, and P. Raghavan. Robust algorithms and software for parallel pde-based simulations. In *Proceedings of the Advanced Simulation Technologies Conference, ASTC*, volume 4, pages 18–22, 2004.

[17] S. Bhowmick, P. Raghavan, L. McInnes, and B. Norris. Faster pde-based simulations using robust composite linear solvers. *Future Generation Computer Systems*, 20(3):373–387, 2004.

[18] S. Bhowmick, P. Raghavan, and K. Teranishi. A combinatorial scheme for developing efficient composite solvers. In *Computational ScienceICCS 2002*, pages 325–334. Springer, 2002.

[19] C. Bielza and P. Larrañaga. Discrete Bayesian Network classifiers: A survey. *ACM Comput. Surv.*, 47(1):5:1–5:43, July 2014.

[20] R. Bramley, D. Gannon, T. Stuckey, J. Villacis, E. Akman, J. Balasubramanian, F. Breg, S. Diwan, and M. Govindaraju. The linear system analyzer. Technical report, Technical Report TR511, Indiana University, 1998.

[21] R. Bramley and X. Wang. Splib: A library of iterative methods for sparse linear systems. *URL address: http://www. elk. itu. edu. tr/dag/lssmc. html*, 1995.

[22] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[23] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[24] P. Cunningham and S. J. Delany. k-nearest neighbour classifiers. *Multiple Classifier Systems*, pages 1–17, 2007.

[25] T. A. Davis. *Direct methods for sparse linear systems*, volume 2. Siam, 2006.

[26] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.

[27] J. Dongarra and V. Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. In *International Conference on Computational Science*, pages 759–767. Springer, 2003.

[28] J. Dongarra and V. Eijkhout. Self-adapting numerical software for next generation applications. *International Journal of High Performance Computing Applications*, 17(2):125–131, 2003.

[29] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. Linpack users guide: Society for industrial and applied mathematics. *Philadelphia, Pennsylvania*, 1979.

[30] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Clarendon press Oxford, 1986.

[31] V. Eijkhout et al. Overview of iterative linear system solver packages. *NHSE review*, 3(1), 1998.

[32] V. Eijkhout and E. Fuentes. Multi-stage learning of linear algebra algorithms. In *Machine Learning and Applications, 2008. ICMLA'08. Seventh International Conference on*, pages 402–407. IEEE, 2008.

[33] V. Eijkhout, E. Fuentes, N. Ramakrishnan, P. Kang, S. Bhowmick, D. Keyes, and Y. Freund. A self-adapting system for linear solver selection. In *Proc. 1st intl workshop on automatic performance tuning (iWAPT2006)*, pages 44–53, 2006.

[34] A. Ern, V. Giovangigli, D. E. Keyes, and M. D. Smooke. Towards polyalgorithmic linear system solvers for nonlinear elliptic problems. *SIAM Journal on Scientific Computing*, 15(3):681–703, 1994.

[35] C. Farhat and F.-X. Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32(6):1205–1227, 1991.

[36] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[37] R. W. Freund, G. H. Golub, and N. M. Nachtigal. Iterative solution of linear systems. *Acta numerica*, 1:57–100, 1992.

[38] R. W. Freund and N. M. Nachtigal. Qmr: a quasi-minimal residual method for non-hermitian linear systems. *Numerische mathematik*, 60(1):315–339, 1991.

[39] W. M. Gentleman. Least squares computations by givens transformations without square roots. *IMA Journal of Applied Mathematics*, 12(3):329–336, 1973.

[40] G. H. Golub and R. S. Varga. Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order richardson iterative methods. *Numerische Mathematik*, 3(1):157–168, 1961.

[41] M. H. Gutknecht and S. Röllin. The chebyshev iteration revisited. *Parallel Computing*, 28(2):263–283, 2002.

[42] A. Hadjidimos. Successive overrelaxation (sor) and related methods. *Journal of Computational and Applied Mathematics*, 123(1):177–199, 2000.

[43] M. A. Heroux and J. M. Willenbring. Trilinos users guide, 2003.

[44] M. R. Hestenes and E. Stiefel. *Methods of conjugate gradients for solving linear systems*, volume 49. NBS, 1952.

[45] G. Holmes, B. Pfahringer, R. Kirkby, E. Frank, and M. Hall. Multiclass alternating decision trees. In *ECML*, pages 161–172. Springer, 2001.

[46] B. Hutchinson and G. Raithby. A multigrid method based on the additive correction strategy. *Numerical Heat Transfer, Part A: Applications*, 9(5):511–537, 1986.

[47] W. Iba and P. Langley. Induction of one-level decision trees. In *Proceedings of the ninth international conference on machine learning*, pages 233–240, 1992.

[48] B. M. Irons. A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2(1):5–32, 1970.

[49] K. Jbilou, A. Messaoudi, and H. Sadok. Global fom and gmres algorithms for matrix equations. *Applied Numerical Mathematics*, 31(1):49–63, 1999.

[50] X. S. Li, J. Demmel, J. Gilbert, L. Grigori, M. Shao, and I. Yamazaki. Superlu numerical software library. *URL http://crd-legacy. lbl. gov/xiaoye/SuperLU*.

[51] M. Manguoglu, F. Saied, A. Sameh, and A. Grama. Performance models for the spike banded linear system solver. *Scientific Programming*, 19(1):13–25, 2011.

[52] L. McInnes, B. Norris, S. Bhowmick, and P. Raghavan. Adaptive sparse linear solvers for implicit cfd using newton-krylov algorithms. In *Proceedings of the Second MIT Conference on Computational Fluid and Solid Mechanics*, volume 2, pages 1024–1028, 2003.

[53] L. C. McInnes, B. Smith, H. Zhang, and R. T. Mills. Hierarchical krylov and nested krylov methods for extreme-scale computing. *Parallel Computing*, 40(1):17–31, 2014.

[54] R. T. Mills, G. E. Hammond, P. C. Lichtner, V. Sripathi, G. K. Mahinthakumar, and B. F. Smith. Modeling subsurface reactive flows using leadership-class computing. In *Journal of Physics: Conference Series*, volume 180, page 012062. IOP Publishing, 2009.

[55] K. Morikuni, L. Reichel, and K. Hayami. Fgmres for linear discrete ill-posed problems. *Applied Numerical Mathematics*, 75:175–187, 2014.

[56] D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear algebra and its applications*, 29:293–322, 1980.

[57] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM journal on numerical analysis*, 12(4):617–629, 1975.

[58] C. C. Paige and M. A. Saunders. Algorithm 583: Lsqr: Sparse linear equations and least squares problems. *ACM Transactions on Mathematical Software (TOMS)*, 8(2):195–209, 1982.

[59] C. C. Paige and M. A. Saunders. Lsqr: An algorithm for sparse linear equations and sparse least squares. *ACM transactions on mathematical software*, 8(1):43–71, 1982.

[60] B. T. Polyak. The conjugate gradient method in extremal problems. *USSR Computational Mathematics and Mathematical Physics*, 9(4):94–112, 1969.

[61] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[62] J. Rice. On the construction of poly-algorithms for automatic numerical analysis. In *Interactive Systems for Experimental Applied Mathematics. M. Klerer and J.Reinfelds*, pages 301–313. Academic Press, 1968.

[63] J. R. Rice. A polyalgorithm for the automatic solution of nonlinear equations. pages 179–183, New York, NY, USA, 1969. ACM.

[64] J. R. Rice. A polyalgorithm for the automatic solution of nonlinear equations. In *Proceedings of the 1969 24th national conference*, pages 179–183. ACM, 1969.

[65] L. Říha, T. Brzobohatỳ, A. Markopoulos, O. Meca, and T. Kozubek. Massively parallel hybrid total feti (htfeti) solver. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, page 7. ACM, 2016.

[66] Y. Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993.

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

[68] Y. Saad and M. H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[69] Y. Saad and B. Suchomel. Arms: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical linear algebra with applications*, 9(5):359–378, 2002.

[70] D. Sculley. Combined regression and ranking. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 979–988. ACM, 2010.

[71] S. S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[72] P. Sonneveld. Cgs, a fast lanczos-type solver for nonsymmetric linear systems. *SIAM journal on scientific and statistical computing*, 10(1):36–52, 1989.

[73] A. Stathopoulos and J. R. McCombs. Primme: preconditioned iterative multimethod eigensolvermethods and software description. *ACM Transactions on Mathematical Software (TOMS)*, 37(2):21, 2010.

[74] J. Todd. *Survey of numerical analysis*. McGraw-Hill, 1962.

[75] L. N. Trefethen and D. Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[76] 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):631–644, 1992.

[77] J. Varah. A survey of iterative methods for sparse linear systems. In *Proceedings of the fifteenth Manitoba conference on numerical mathematics and computing (Winnipeg, Man., 1985)*, pages 83–92, 1986.

[78] E. W. Weisstein. Biconjugate gradient method. 2003.