# Development of Scalable Efficient Solvers for Sparse Linear System of Equations

*Project report submitted*

*In fulfilment of the requirement for the degree of*

*Bachelor of Technology*

*By*

*Anshul Goyal* (130103011)

*Kanishk Chaturvedi* (130103035)

**Department of Mechanical Engineering**

**Indian Institute of Technology Guwahati**

**November, 2016**

# CERTIFICATE

It is certified that the work contained in the project report titled **"Development of Scalable Efficient Solvers for Sparse Linear System of Equations"**, by **Anshul Goyal** (130103011) and **Kanishk Chaturvedi** (130103035) has been carried out under our supervision and that this work has not been submitted elsewhere for the award of a degree.

Dr. Deepak Sharma

November, 2016 Department of Mechanical Engineering,

I.I.T. Guwahati.

Dr. S. S. Gautam

November, 2016 Department of Mechanical Engineering,

I.I.T. Guwahati.

# DECLARATION

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

**Anshul Goyal (130103011)**

**Kanishk Chaturvedi (130103035)**

**7 November, 2016**

# ACKNOWLEDGEMENT

We feel a great privilege in expressing our deepest and most sincere gratitude to our supervisor, Dr. Deepak Sharma and Dr. S. S. Gautam for the most valuable guidance and influential mentorship provided to us during the course of this project. Due to his technical advices, exemplary guidance, persistent monitoring and constant encouragement throughout the course of our project work, we were able to complete the project through various stages.

Anshul Goyal
Department of Mechanical Engineering
I.I.T. Guwahati.

Kanishk Chaturvedi
Department of Mechanical Engineering,
I.I.T. Guwahati.

# ABSTRACT

The modern era is continuously advancing in the sphere of technology, hence, posing even more complex problems with each step towards a more scientific world. The computations that were earlier possible through direct methods have grown into much larger problems with higher time and space complexities. This calls for a need of further optimization of the pre-existent direct solvers. Hence, certain Iterative solvers like Conjugate Gradient (CG) Method are developed to further satisfy the memory and time constraints of each complex problem.

Although such solvers have experienced a kind of maturation over the past few years, this relatively new sphere of development of Sparse System Solvers is still vulnerable to many undone optimizations and improvements in each particular iteration. If the physical aspects and properties of a mechanical system are exploited in a careful manner, we could reach an even more efficient method of performing calculations on sparse real world problems.

Furthermore, the iterative methods are far more easier to implement on parallel systems like GPU, which helps us to explore the various options of custom modifying the algorithms according to our physical constraints, in order  to advance its performance.

Keywords: Sparse Systems, Solvers, Conjugate Gradient, Time & Space Complexities

# NOMENCLATURE

**A -** Stiffness Matrix

**x, y, z -** Solution Vectors

**B, b -** Right Hand Vectors

**L -** Lower Triangular Matrix

**U -** Upper Triangular Matrix

**u, v -** Row Vectors

**m -** number of rows in the matrix **A**

**M -** an arbit **m x m** matrix

**nnz -** number of nonzero elements

**val, row_ptr, col_ind -** Arrays used in CSR format

**$p_i$ -** direction vectors

**$r_k$ -** Residual Vector

$\alpha$, $\beta$, k - constants

**R,Z -** Column Vectors

# CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# Chapter 1
# INTRODUCTION

## 1.1 Introduction to Solvers of system of Linear Equations

We face various kinds of challenges on technical level in day-to-day life. As we are moving ahead in this wormhole, we are encountering new discoveries and new challenges. These challenges involve new problems as well which we intend to overcome. Mathematics as we know is the mother of all inventions and without it, we simply cannot prove or justify anything, and here is where it becomes useful. We generally try and divide a problem into sets of equations involving variables which we do not know and which we need to figure out. As the number of equations increase it becomes tedious to write it all downs, thus we came up with matrices. A basic set of system of Linear Equation is written as,

$$\mathbf{Ax = b}$$

Now, A and b are the matrix of coefficient and right hand side (RHS) values respectively, while x is the solution matrix or the matrix of variables which we need to find out. The methods of solving and finding out this **x** are broadly termed as **"Solvers".**

There are two broad classification of Solvers,

1) Direct Solvers
2) Iterative Solvers

Direct solvers are those compute the solution to a problem in a finite number of steps.

Iterative methods are not expected to end in a finite number of steps, they rather involve successive approximations that converge to the exact solution.

## 1.2 Motivation [5]

Finding trends in similar kinds of problems we can easily categorize sets of linear equations. The bigger the problem, more are the number of equations and more is the complexity involved, and nowadays no-thing is small. This as well results in significant increase in computation and time. Thus we intend to utilize technique of divide and conquer to do our job. This can be achieved with the help of parallel computing. Parallelism is a novel field and is rising day by day, because of its ability to divide a bigger problem in smaller chunks and run many computations at a time, thus reducing the total time of computation significantly. We as youngsters of this generation believe in this method and want to carry the baton to further fields in newer ways.

## 1.3 Objectives

1) Implementation of Solver on CPU (Central Processing Unit).

2) Comparison of pre-existing library with our own implementation in CPU.

3) Implementation of solver on GPU (Graphics Processing Unit).

4) Comparison of pre-existing library with our own implementation in GPU.

## 1.4 Organization of report

The reports is organized in five chapters. Chapter 2 presents literature survey on existing solver for system of linear equations. Chapter 3 presents implementation of the conjugate gradient method on CPU. The results and discussion are presented in Chapter 4. The report is concluded in Chapter 5 with scope of future work for BTP Phase-II.

# Chapter 2

# LITERATURE REVIEW

## 2.1 Direct Solvers [1]

Direct solvers are those compute the solution to a problem in a finite number of steps.

### 2.1.1 Gaussian Elimination [1]

To solve the linear system of equations **Ax = B**, one of the popular methods of computation has been the Direct Solver in the form of **Gaussian Elimination**. The Gaussian Elimination (GE) procedure primarily involves two steps of computation:

1) Decomposing the matrix **A** into the product of two matrices: **L** and **U** matrices.
2) Solving **Ly = B** and **Uz = y** by forward and backward substitutions respectively.

The above procedure, as a result, had to store **L** and **U** matrices for carrying out the respective substitutions and hence, comprised of a storage requirement of order of square of **m**.

### 2.1.2 Gauss Jordan Elimination [1]

The following method involves a set of row operations to change the augmented matrix of a system to an Identity Matrix. The allowed steps are,

1. Interchange any two rows.
2. Multiply each element of a row by a nonzero constant.
3. Replace a row by the sum of itself and a constant multiple of another row of the matrix.

## 2.2 Iterative Solvers [1]

Iterative methods do not end in a finite number of steps, they rather involve successive approximations that converge to the exact solution. Few of the Iterative solvers are,

### 2.2.1 Gradient Descent Method [1]

**Gradient descent** is a first-order iterative optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient of the function at the current point. For a function multi variable function F(x), the steepest or fastest decrease is along the direction of the gradient at any point at which the function is differentiable.

### 2.2.2 Conjugate Gradient Method [1]

Conjugate Gradient is also a type of Iterative solver which gives us the solution of the scenario where the system of linear equations have a matrix which is symmetric and is positive definite. We have chosen this method as our base for all solution formation and calculations, this method is discussed in detail later in the report.

## 2.3 Scope and Solver Selection [1]

The fallout of such an approach was that for matrices of a greater size(such as order of $10^9$) , the computations tend to become redundant and slower due to large storage spaces required even for the zeroes present in the matrix, and hence, the huge sizes of the matrix **A**.
Hence, to cope up with the real time and space constraints and maintain the efficiency of our solver, we settled on the Iterative Conjugate Gradient Method (CG) as the best approach to tackle such a problem involving multiple computations with large Sparse Matrices having most of its elements as zeroes. Moreover, our coefficient matrix **A** is symmetric and positive definite, hence making it possible to utilize the CG approach.

With regards to the time complexity among the various techniques, we observed that the CG method is better than every other optimal steepest descent method as the former converges to a solution point faster with every iteration of the algorithm, owing to the utilization of the scalar variable $\beta$ (elaborated later) which improves the direction of convergence of the solution after every iteration in the CG technique.

The iterative CG method is the best for implementation of our future algorithm on GPU parallel systems as they are inexpensive in memory and much time efficient in computations than the direct solvers similar to GE. This was made possible by the incorporation of a sparse storage technique known as Compressed Row Storage (CRS) [3] method into our Conjugate Gradient algorithm.

The most important measure of the efficiency of our algorithm has been the comparison with the standard & recognized library for employing CG method in sparse matrices, namely: **Eigen** [4].

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.

# Chapter 3
# IMPLEMENTATION OF CONJUGATE GRADIENT METHOD IN CPU

As discussed above, we have deduced why we have chosen **Conjugate Gradient** [2] or **CG** method. In this section we will discuss how did we implement the following method and ran on CPU without any parallelism involved.

Conjugate Gradient Method as discussed above is often implemented as an iterative algorithm where we derive subsequent results from the previous ones.

## 3.1 Conjugate Gradient Method

Just to start with an overview, if we want to solve a system of linear equations,

$$\mathbf{A\ x=b}$$

for the vector x where the known $n \times n$ matrix **A** is **symmetric**, **positive definite**, and **real**, and **b** is known as well. We denote the unique solution of this system by **x**.

### 3.1.1 Initial Definitions

Now before entering into the actual algorithm, few terms are to be defined here,

*A **symmetric matrix** is a square **matrix** that is equal to its transpose.*

$$\mathbf{A^T=A}$$

*A positive definite matrix is a symmetric matrix with all positive eigenvalues.*

$$\mathbf{uAu^T}>0$$

*Two non-zero vectors are said to be conjugate with respect to* **A** *if,*

$$\mathbf{u^TAv}=0$$

From the above, since **A** is symmetric and positive definite, the left-hand side defines an inner product.

$$\mathbf{<u,v>_A := <Au,v> = <u,A^Tv> = <u,Av> = uA^Tv}$$

Two vectors are conjugate if and only if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if **u** is conjugate to **v**, then **v** is conjugate to **u**.
Now that we have defined the necessary terms we will first know how the solution sets depends upon a set of particular directions which are mutually conjugate to each other.

## 3.1.2 Conjugate gradient as a Direct Method

Let us suppose that P={$\mathbf{p_1}, \mathbf{p_2}, \dots, \mathbf{p_n}$} are mutually conjugate vectors.Now they form the basis for $R_n$, and we may express the solution $\mathbf{x}_*$ of **A x = b** in this basis :

$$\mathbf{x_*}=\sum_{i=1}^{n} \alpha i\ \mathbf{pi}$$

Based on this expansion we calculate:

$$\mathbf{Ax_*}=\sum_{i=1}^{n} \alpha\ \mathbf{Api}$$

$$\mathbf{p_k}^T\mathbf{Ax_*}=\sum_{i=1}^{n} \quad \alpha_i\mathbf{p_k}^T\ \mathbf{Ap_i}$$

$$\mathbf{p_k}^T\mathbf{b}=\sum_{i=1}^{n} \quad \alpha_i<\mathbf{p_k},\mathbf{p}>_A$$

$$<\mathbf{p_k},\mathbf{b}>=\alpha_k<\mathbf{p_k},\mathbf{p}>_A$$

$$\alpha_k=<\mathbf{p_k},\mathbf{b}> / <\mathbf{p_k},\mathbf{p}>_A$$

This gives the following method for solving the equation **Ax** = **b**: find a sequence of *n* conjugate directions, and then compute the coefficients $\alpha_k$.

## 3.1.2 Conjugate gradient as an Iterative Method

If we choose the conjugate vectors $\mathbf{p}_k$ carefully, then we may not need all of them to obtain a good approximation to the solution $\mathbf{x}_*$. So, we want to regard the conjugate gradient method as an iterative method.

We denote the initial guess for $\mathbf{x}_*$by $\mathbf{x}_0$. We can assume without loss of generality that $\mathbf{x}_0 = \mathbf{0}$. Starting with $\mathbf{x}_0$ we search for the solution and in each iteration we need a metric to tell us whether we are closer to the solution $\mathbf{x}_*$ (that is unknown to us). This metric comes from the fact that the solution $\mathbf{x}_*$ is also the unique minimizer of the following quadratic function;

$$\mathbf{f(x) = x^TAx\text{-}x^Tb}$$

This suggests taking the first basis vector $p_0$ to be the negative of the gradient of **f(x)** at $x = x_0$. The gradient of **f(x)** equals $\mathbf{Ax - b}$. Starting with a "guessed solution" $x_0$ (we can always guess $x_0 = 0$), this means we take $\mathbf{p_0 = b - Ax_0}$. The other vectors in the basis will be conjugate to the gradient, hence the name *conjugate gradient method*. Let $\mathbf{r}_k$ be the residual at the *k*th step:

$$\mathbf{r_k=b\text{-}Ax_k}$$

Note that $r_k$ is the negative gradient of **f(x)** at $x = x_k$, so the gradient descent method would be to move in the direction $r_k$. Here, we insist that the directions $p_k$ be conjugate to each other.

We also require that the next search direction be built out of the current residue and all previous search directions, which is reasonable enough in practice.

This results in the following expression,

$$\mathbf{p_k = r_k} - \sum_{i<k}^{i=1} (\mathbf{p_i}^T \mathbf{A r_k} / \mathbf{p_i}^T \mathbf{A p_i}) * \mathbf{p_i}$$

Following this direction, the next optimal location is given by,

$$\mathbf{x_{k+1} = x_k} + \alpha_k \mathbf{p_k} \text{ , and}$$

$$\alpha_k = (\mathbf{p_k}^T \mathbf{r_{k-1}}) / (\mathbf{p_k}^T \mathbf{A p_k})$$

A closer analysis of the algorithm shows that $r_{k+1}$ is orthogonal to $p_i$ for all $i < k$ , and therefore only $r_k$, $p_k$, and $x_k$ are needed to construct $r_{k+1}$, $p_{k+1}$, and $x_{k+1}$.

## 3.2 Pseudo CPU code for CG

Thus the final algorithm is as follows,

```
$\mathbf{r_0 := b - Ax_0}$
$\mathbf{p_0 := r_0}$
k :=0
repeat
    $\alpha_k := (\mathbf{r_k}^T \mathbf{r_k}) / (\mathbf{p_k}^T \mathbf{A p_k})$
    $\mathbf{x_{k+1} = x_k} + \alpha_k \mathbf{p_k}$
    $\mathbf{r_{k+1} = r_k} - \alpha_k \mathbf{A p_k}$
    if $r_{k+1}$ is sufficiently small then exit loop
    $\beta_k = (\mathbf{r_{k+1}}^T \mathbf{r_{k+1}}) / (\mathbf{r_k}^T \mathbf{r_k})$
$\mathbf{p_{k+1} = r_{k+1}} + \beta_k \mathbf{p_k}$
    k=k+1
end repeat
The result is $\mathbf{x_{k+1}}$
```

## 3.3 Compressed sparse row format

The **Compressed Sparse Row** (CSR) format represents a matrix **M** by three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices. This format allows fast row access and matrix-vector multiplications (M*y*). The CSR format stores a sparse matrix **M** in row form using three (one-dimensional) arrays namely **"val, row_ptr, col_ind"**. Let **"nnz"** denote the number of nonzero entries in **M**.

The array **"val"** is of length **"nnz"** and holds all the nonzero entries of **M** in left-to-right top-to-bottom ("row-major") order.

The array **"row_ptr"** is of length ***m*** + 1. It is defined by this recursive definition:

- **row_ptr**[0] = 0
- **row_ptr**[*i*] = **row_ptr**[*i* − 1] + number of nonzero elements on the (*i* − 1)−th row in the original matrix
- Thus, the first ***m*** elements of **row_ptr** store the index into **val** of the first nonzero element in each row of **M**, and the last element **row_ptr**[***m***] stores **nnz**, the number of elements in **val**.

The third array, **col_ind**, contains the column index in **M** of each element of **val** and hence is of length **nnz** as well.

Using the following format of storage of sparse matrices, we were able to reduce the total number of multiplications to a large extent, as this particular method takes care of neglecting the multiplications done with **zero-elements** present in the stiffness matrix, thus reducing the time and computation.

The algorithm used for multiplying matrices stored in CSR format is as below:

```
Initialize Vector R to {0}
m = number of rows in Matrix A
Iterate i from 0 to m
    Iterate j from row_ptr[i] to row_ptr[i+1]
        R[i,0]=R[i,0]+val[j]*Z[col_ind[j],0]
```

# Chapter 4

# RESULTS AND DISCUSSIOn

## 4.1 Problem Statement

For the primitive testing of our code and implementation we chose a simple two dimensional Thermal Conduction problem, where Temperature is one and only index which we assume to change. Thus it makes it a one-degree of freedom problem which is quite easy to solve using the concepts of FEM.

The Problems Parameters are as follows,

1) Problem consists of a square shaped two-dimensional body, where we can manually specify the division of nodes and elements as per our will
2) The scenario is a One-Degree of Freedom problem where each element is a four noded quadrilateral.

The Boundary Conditions are as follows,

1) Constant heat flux of 1 W/m$^2$ at top and bottom edge of square body.
2) Constant Temperature of 1$^0$C at the left edge of square body.
3) Insulated boundary condition at right edge of square body.
4) No Heat Generation per unit volume involved.

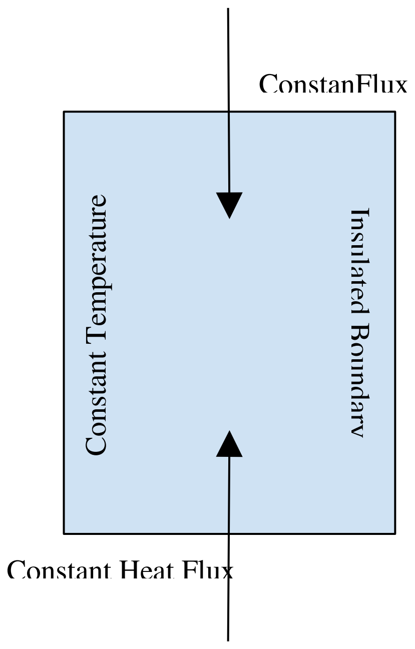


Figure 4.1: Schematic of the Problem Statement along with boundary conditions

The following problem was picked, as because it had less complexity involved and was perfect to start with. Us being new to this field required something which was less typical and easy to compare and run our code, and thus this was selected.

We had access to a MATLAB code for the following problem which helped us to generate solutions for the desired number of elements. The number of elements could be easily changed inside the code and subsequently run to get the solution for it. The code also generates the stiffness matrix and the right hand matrix for the subsequent iterations. We generated many such combinations and ran these different cases for our code. Our code generated answers with high precision and optimum time which we again compared with a solution from the pre-existing library **Eigen.** We will be exploring the results and comparisons later in this report.

## 4.2 Experimental Data Obtained

The following table of data has the following parameters:

Column 1: Number of Elements in the Sparse Coefficient Matrix in one orthogonal direction, which varies from 2 to 60 in our experimental data.

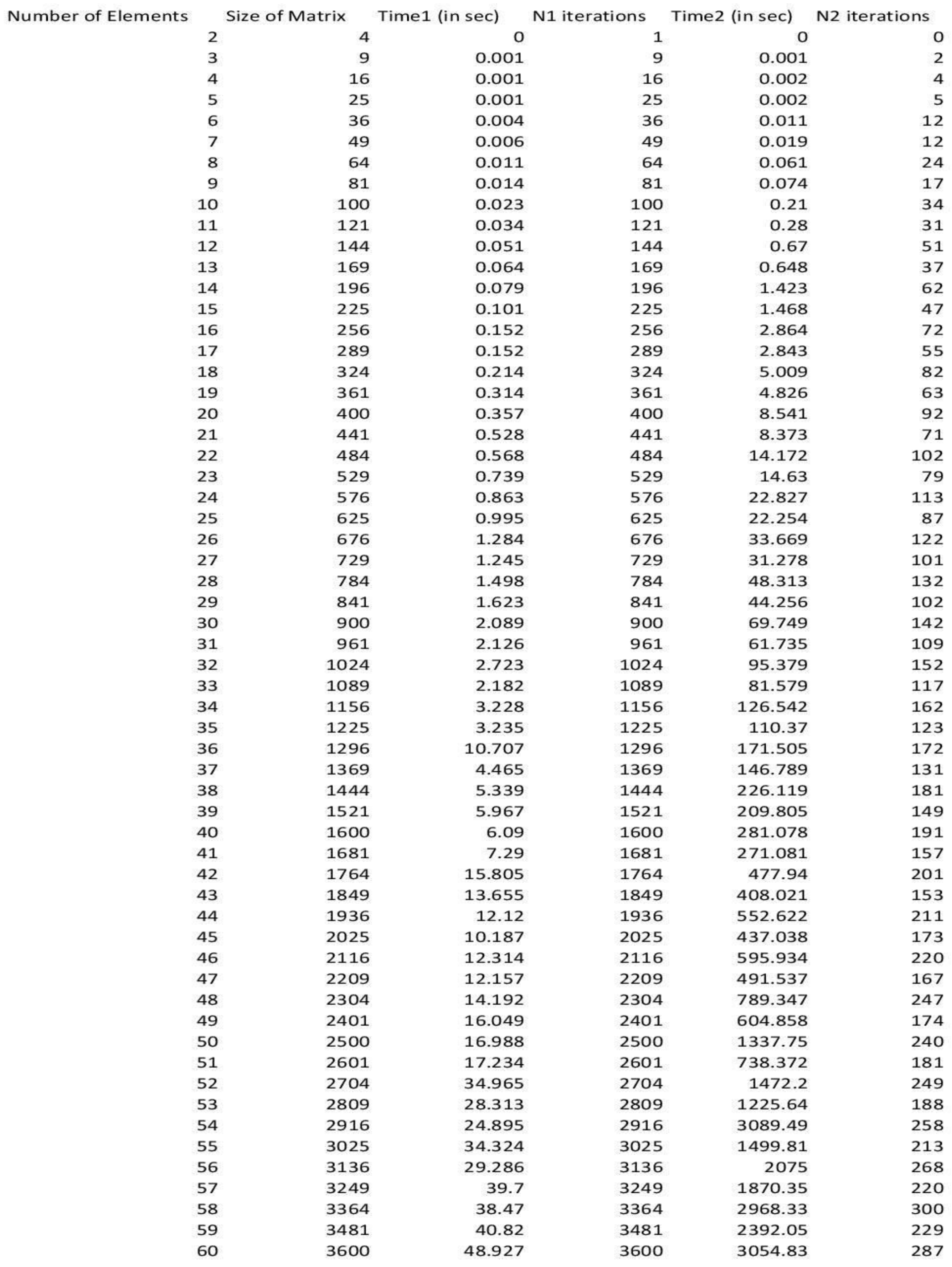
Column 2: Size of the coefficient matrix which is (N X N), where N varies from (4 to 3600).

Column 3: Time (T1) in seconds required for the developed solver to achieve the results for a given matrix size n.

Column 4: Number of Iterations N1 of the developed algorithm to achieve the final solution with optimum accuracy for a given input matrix size N.

Column 5: Time (T2) in seconds required for the standard Eigen solver to achieve the results for a given matrix size N.

Column 6: Number of Iterations N2 of the standard Eigen algorithm to achieve the final solution for a given input matrix size n.



| Number of Elements | Size of Matrix | Time1 (in sec) | N1 iterations | Time2 (in sec) | N2 iterations |
|---|---|---|---|---|---|
| 2 | 4 | 0 | 1 | 0 | 0 |
| 3 | 9 | 0.001 | 9 | 0.001 | 2 |
| 4 | 16 | 0.001 | 16 | 0.002 | 4 |
| 5 | 25 | 0.001 | 25 | 0.002 | 5 |
| 6 | 36 | 0.004 | 36 | 0.011 | 12 |
| 7 | 49 | 0.006 | 49 | 0.019 | 12 |
| 8 | 64 | 0.011 | 64 | 0.061 | 24 |
| 9 | 81 | 0.014 | 81 | 0.074 | 17 |
| 10 | 100 | 0.023 | 100 | 0.21 | 34 |
| 11 | 121 | 0.034 | 121 | 0.28 | 31 |
| 12 | 144 | 0.051 | 144 | 0.67 | 51 |
| 13 | 169 | 0.064 | 169 | 0.648 | 37 |
| 14 | 196 | 0.079 | 196 | 1.423 | 62 |
| 15 | 225 | 0.101 | 225 | 1.468 | 47 |
| 16 | 256 | 0.152 | 256 | 2.864 | 72 |
| 17 | 289 | 0.152 | 289 | 2.843 | 55 |
| 18 | 324 | 0.214 | 324 | 5.009 | 82 |
| 19 | 361 | 0.314 | 361 | 4.826 | 63 |
| 20 | 400 | 0.357 | 400 | 8.541 | 92 |
| 21 | 441 | 0.528 | 441 | 8.373 | 71 |
| 22 | 484 | 0.568 | 484 | 14.172 | 102 |
| 23 | 529 | 0.739 | 529 | 14.63 | 79 |
| 24 | 576 | 0.863 | 576 | 22.827 | 113 |
| 25 | 625 | 0.995 | 625 | 22.254 | 87 |
| 26 | 676 | 1.284 | 676 | 33.669 | 122 |
| 27 | 729 | 1.245 | 729 | 31.278 | 101 |
| 28 | 784 | 1.498 | 784 | 48.313 | 132 |
| 29 | 841 | 1.623 | 841 | 44.256 | 102 |
| 30 | 900 | 2.089 | 900 | 69.749 | 142 |
| 31 | 961 | 2.126 | 961 | 61.735 | 109 |
| 32 | 1024 | 2.723 | 1024 | 95.379 | 152 |
| 33 | 1089 | 2.182 | 1089 | 81.579 | 117 |
| 34 | 1156 | 3.228 | 1156 | 126.542 | 162 |
| 35 | 1225 | 3.235 | 1225 | 110.37 | 123 |
| 36 | 1296 | 10.707 | 1296 | 171.505 | 172 |
| 37 | 1369 | 4.465 | 1369 | 146.789 | 131 |
| 38 | 1444 | 5.339 | 1444 | 226.119 | 181 |
| 39 | 1521 | 5.967 | 1521 | 209.805 | 149 |
| 40 | 1600 | 6.09 | 1600 | 281.078 | 191 |
| 41 | 1681 | 7.29 | 1681 | 271.081 | 157 |
| 42 | 1764 | 15.805 | 1764 | 477.94 | 201 |
| 43 | 1849 | 13.655 | 1849 | 408.021 | 153 |
| 44 | 1936 | 12.12 | 1936 | 552.622 | 211 |
| 45 | 2025 | 10.187 | 2025 | 437.038 | 173 |
| 46 | 2116 | 12.314 | 2116 | 595.934 | 220 |
| 47 | 2209 | 12.157 | 2209 | 491.537 | 167 |
| 48 | 2304 | 14.192 | 2304 | 789.347 | 247 |
| 49 | 2401 | 16.049 | 2401 | 604.858 | 174 |
| 50 | 2500 | 16.988 | 2500 | 1337.75 | 240 |
| 51 | 2601 | 17.234 | 2601 | 738.372 | 181 |
| 52 | 2704 | 34.965 | 2704 | 1472.2 | 249 |
| 53 | 2809 | 28.313 | 2809 | 1225.64 | 188 |
| 54 | 2916 | 24.895 | 2916 | 3089.49 | 258 |
| 55 | 3025 | 34.324 | 3025 | 1499.81 | 213 |
| 56 | 3136 | 29.286 | 3136 | 2075 | 268 |
| 57 | 3249 | 39.7 | 3249 | 1870.35 | 220 |
| 58 | 3364 | 38.47 | 3364 | 2968.33 | 300 |
| 59 | 3481 | 40.82 | 3481 | 2392.05 | 229 |
| 60 | 3600 | 48.927 | 3600 | 3054.83 | 287 |

Table 4.1: Observation table consisting of both the results of our implementation and that of Library for comparison.

## 4.3 Simulation Results

Keep in mind that in the following figures, the color Blue is corresponding to our developed Scalable Solver whereas the color Red is corresponding to the standard Eigen Solver.

The following are the graphs obtained for the experimental data along with their individual observations and justifications:

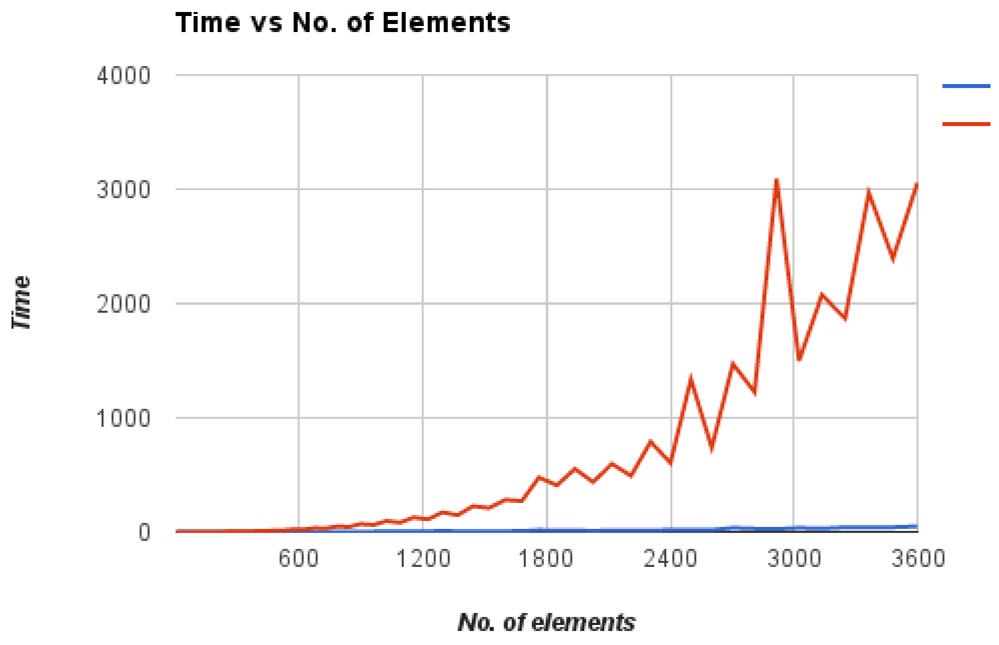


Figure 4.2: Computation time of solvers with respect to the no. of elements

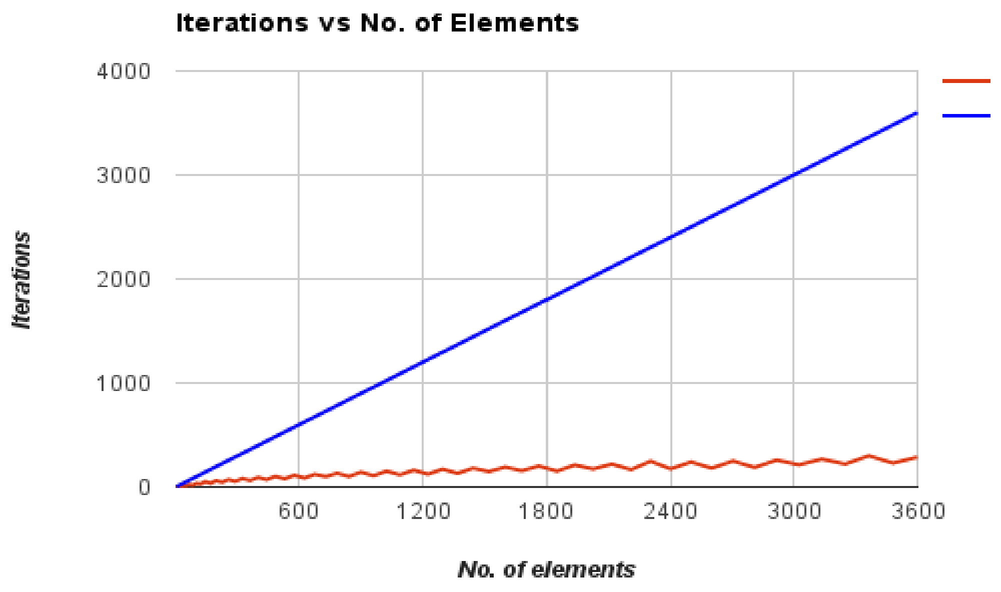


Figure 4.3: No. of iterations of solver with respect to the no. of elements

## 4.4 Observations & Justifications

The following observations were made from the following set of figures:

***Figure 4.2:***

We can observe from the Figure 1 that the computation time of both the solvers increases with increase in input size matrix, the rate of increase of time complexity being much higher for the standard algorithm(red) as compared to the developed solver. This is due to the multiple optimizations done in our algorithm for both the computations and the storage of the data obtained, owing to our technique of considering only the finite (non-zero) values for calculations.

***Figure 4.3:***

One of the observations made from Figure 2 was that the the number of iterations required to reach a final solution increase for both the solvers with increasing matrix input size. This is due to a higher number of iterations needed to minimize the residue of the system by descending to the final solution point in the correct direction in the CG method. With increasing space complexity of problem coefficient matrix, the need for greater number of iterations of the CG algorithm arises to maintain accuracy of the iterative solution. We further observe that the number of iterations in the scalable developed algorithm (blue) is much higher than the standard

Eigen algorithm because the standard CG procedure converges by a greater amount in each iteration owing to its utilization of a preconditioner before the initialization of the computational process.

The maximum limit of iterations of our solver is fixed at the value corresponding to the size of our coefficient matrix, hence we obtain a linear blue graph for matrices of greater sizes in the given range, and we manage to maintain accuracy till four places of decimal for every finite numerical element even when we limit the number of iterations proportionally to the matrix size.

# Chapter 5
# Conclusions and Future Work

## 5.1 Conclusions

From the obtained graphical data and analysis, we can construe the following experimental conclusions:

1) As observed from Figure 1, the time complexity of both the developed solver and the pre-existent standard Eigen solver increases with the increase in the input size of the sparse coefficient matrix A.

2) From Figure 1, comparing the time complexities of the two, we conclude that the time complexity of our algorithm is better than that of the standard Eigen Library by a factor of order hundred or more for the whole input range varying from SPD (symmetric positive definite) matrices of size (4X4) to (3600X3600).

3) The sparse storage CRS format consists of 3 sparse storage vectors which greatly optimizes the space complexity of our algorithm, corresponding to the **eigen::makeCompressed()** command in the standard library.The conclusion (2) is true despite the fact that the number of iterations performed by our algorithm to reach the end result is far greater(usually by a factor of 100) than the standard library algorithm. It is worth noting that the number of iterations for both the procedures increase with the increase in the input size, in order to reach the final minimum residue solution.

4) Hence, we comfortably manage to maintain the accuracy of our results (cross checked by the results from MATLAB solver and the solutions from the library) within the recommended time and space constraints. Thus, we can finally reach the conclusion that the CG solver developed in our project work achieves the result for the given initial problem maintaining the accuracy till four places of decimal for every numerical element in the solution, and that too within much better time complexity and utilization of greater number of iterations for every input matrix of size(**m x m**), where **m** varies from 4 to 3600 in the experimental data.

5) The much better performance of our solver algorithm within the constraints of the initial value problem can be owed to the multiple optimizations done at each mathematical and storage operation within the iterative technique (for instance, the vector multiplication of sparse matrices), which are handled by our own custom developed optimized functions to carry out particular operations in the solver.

## 5.2 Future Work for BTP Phase-II

Keeping in mind the above drawn conclusions, we can thoroughly map our path to future perspectives such as computation in GPU parallel systems utilizing the Parallel Programming and Computing Platform CUDA . CUDA is a parallel computing platform and application programming interface (API) model created by Nvidia.

Such parallel computations on each elementary step of our CG method greatly improves the time complexity from our benchmark of CPU computations performed till now, and corresponding to our competition with the Eigen Library in CPU systems, puts us in place to further compute even with the more robust Cusp library for GPU parallel systems.Cusp is a library for sparse linear algebra and graph computations and it provides a flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems.

Furthermore, as a future objective, we aim to optimize the CG method according to the parallel computational environment using a particular system of Kernels of synchronous threads depending on the space and time constraints of the calculation, multiplying it with the optimizations of the storage technique CRS(compressed row storage).

# References

[1] Press, W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P., “Numerical Recipies in C++”, 2nd ed., Cambridge University Press, 2002.

[2] “Conjugate Gradient-Wikipedia” <en.wikipedia.org/wiki/Conjugate_gradient_method>, July 2016.

[3] “Compressed Row Format-Wikipedia” <en.wikipedia.org/wiki/Sparse_matrix>, July 2016.

[4] Eigen Sparse Matrix Library, C++ <eigen.tuxfamily.org/dox/index.html>, July 2016

[5] Bell, N., Garland, M., “Efficient Sparse Matrix-Vector Multiplication on CUDA”, December 11, 2008.

[6] Smailbegovic, F., Gaydadjiev, G. N., Vassiliadis, S., “Sparse matrix storage format”, January 2005.