Parallel Preconditioners for Finite Element Computations

Emily Furst
College of Saint Benedict/Saint John's University

Parallel Preconditioners for Finite Element Computations

AN HONORS THESIS

College of Saint Benedict/Saint John's University

In Partial Fulfillment

of the Requirements for Distinction

In the Departments of *Computer Science* and *Mathematics*

by Emily Furst

April 2015

PROJECT TITLE: *Parallel Preconditioners for Finite Element Computations*

Approved by:

Michael Heroux

Scientist in Residence of Computer Science

Imad Rahal

Associate Professor of Computer Science

Robert Hesse

Associate Professor of Mathematics

Imad Rahal

Chair, Department of Computer Science

Robert Hesse

Chair, Department of Mathematics

Emily Esch

Director, Honors Thesis Program

Abstract

This thesis sought to explore numerical methods for solving partial differential equations and to determine the best method of updating the deal.II software to utilize new Trilinos software packages. The one dimensional heat equation with Dirichlet boundary conditions and nonzero initial conditions was solved analytically, using the Forward in Time, Central in Space scheme of the finite difference method, and the Crank-Nicolson scheme of the finite element method. The solutions from using the finite difference method and the finite element method were then compared to the analytic solution to determine accuracy. An example using the same Trilinos packages that are utilized in deal.II currently was updated to use the newer Trilinos packages to determine how to update deal.II and to analyze any performance increases resulting from these changes to the software.

Contents

1 Introduction

Many of the problems that are of interest to scientists regarding the physical world can best be described by a partial differential equation. Heat dynamics, fluid dynamics, and quantum mechanics can all be described in terms of partial differential equations. However, partial differential equations can be quit difficult and time-consuming to solve analytically. Here, solving analytically refers to using algebraic or numeric methods to find a solution to a partial differential equation where the solution is in the form of an equation. As a result, many softwares have been developed in order to make the solving of such problems easier. However, as computers cannot process the infinite number of points comprising a problem space, finite numerical algorithms were developed to replace traditional analytical methods. These computational methods vary in terms of efficiency and accuracy.

One particular application of computational methods is in the solving of very large problems. This desire to solve exceptionally large problem sizes has led to the development of high performance and parallel methods of solving. Many numerical methods involve quite a bit of repetition in computation which enables the programmer to exploit quite a bit of parallelism. Well thought out and smart storage patterns of data in memory can also be utilized to improve performance. When parallel and high performance methods are implemented well, increased time performance is often noted in the computation time associated with solving such large problems.

1.1 Introduction to Partial Differential Equations and the One Dimensional Heat Equation

Partial differential equations (PDEs) are highly important tools for the understanding of the physical world, and in fact were formulated by scientists in the eighteenth and nineteenth centuries in order to describe different aspects of the physical world [11]. It is important to note that as a result of the motivation for their development and as a result of their nature, it is difficult to think of partial differential equations in an abstract mathematical sense. That is, due to ambiguities and contradictions in solutions, it is often necessary to return the problem to the physical situation it is taken from in order to arrive upon the intended solution and remove ambiguities [11]. An elementary, homogeneous, linear partial differential equation is the one dimensional heat equation first derived by Joseph Fourier in 1807 [11].

When solved, the one dimensional heat equation models the heat distribution across a rod of certain length over time. It was originally stated by Fourier in *Théorie Analytique de la Chaleur* as

$$\boldsymbol{q} = -k \bigtriangledown u$$

where $\boldsymbol{q}$ is the rate of flow of heat energy per unit time, with the conductivity, k a positive constant, and $\bigtriangledown u$ is the gradient of u, the temperature [3]. Heat equations can model the behaviour of a variety of dimensional spaces, materials, and conditions including sources and sinks. For the one dimensional heat equation, it is assumed that the surface of the rod is insulated so that heat can only flow in the x-direction. In connection with their application to physical situations, heat equations can be described by either initial conditions, boundary conditions, or both in order to better describe the situation. Initial conditions describe the initial state of the problem space in terms of some equation or constants. Boundary conditions describe the state of the boundary or edges of the problem space. Depending on how these conditions are specified, they can be classified as being of several different forms. Dirichlet conditions are one such form. Specifically, Dirichlet boundary conditions impose a fixed value for the solution at the ends of the rods. For example, the Dirichlet boundary condition

$$u(0, t) = 0 = u(L, t)$$

would imply that at the ends of the rod, 0 and L, the temperature is fixed at 0 degrees in the solution, u [3]. Once adequately defined, the one dimensional heat equation can then be solved in a variety of methods not only limited to analytical methods.

1.2 Introduction to Parallel and Distributed Computing

As opposed to traditional sequential computing, parallel computing takes advantage of computer architectures in which there are multiple processors. One of the goals of parallel computing is to increase performance by decreasing computation time. This is achieved by separating independent tasks to be computed on different processors at the same time. By computing tasks at the same time the total computation time will typically be less than the total computation time resulting from computing the tasks in serial. Independent tasks are ones that do not rely upon the output of any other task for its own computation.

When multiple machines are being used together and communicating with one another in order to complete a set of computations or run a program, it is referred to as distributed computing. Distributed computing allows for very large problems to be solved using computational methods. By using a distributed system, there is much more memory and computation power available to the user. Scientists trying to solve large partial differential equations often utilize distributed systems in order to be able to solve such large equations and to decrease the length of time that is required to solve the PDE.

2 Contributions to this Topic

This thesis provides several contributions to the field. First, it provides a comprehensive tutorial on solving partial differential equations both analytically and finitely. It takes a single one dimensional heat equation as an example and proceeds to walk through the steps to solve the equation three different ways. An analytical solution is initially found. Next, a simple finite method, finite difference, is chosen to introduce finite solutions. Finally, a more robust finite method, finite element, is chosen to show the advantages of approximate solutions. By solving the same equation for each method, it is easy to see the advantages and disadvantages of each method. In addition, a method for updating the deal.II software is designed and tested in this thesis. There is a desire to update deal.II, and no method had previously been proposed. The method proposed here can serve as a basis for others to continue the work on deal.II. While slight modifications can be made to improve the proposed method, the information gathered will help others to easily pick up and continue the work.

3 Problem Statement

There are many software packages available to aid in a variety of different tasks. However, as hardware advances, oftentimes software becomes outdated and loses its value. Because of this, software is referred to as being ”brittle”. Thus, maintenance of code is becoming an important area of research within computing. Maintenance of code involves updating software to utilize the resources made available by new and advanced computer architectures and programming languages. By practicing code maintenance, the amount of work on programmers and developers is decreased as updating software to use new resources is typically less complex and time consuming than scrapping outdated software and starting from scratch.

In this thesis, we seek to determine an approach to maintain the deal.II software for finite element computations by updating the Trilinos packages that it utilizes. Currently, through wrapper classes, deal.II supports Epetra [1] and Tpetra [1] for matrix and vector representation, ML [1] for matrix preconditioners, and AztecOO [1] for solvers. Our aim is to determine how best to modify the deal.II wrapper classes so that they use the MueLu [1] package for preconditioners and the Belos [1] package for solvers. However, as MueLu only supports Xpetra [1] objects, it is necessary to develop a method to wrap the Epetra and Tpetra objects as Xpetra objects before setting up a MueLu preconditioner and Belos solver. Small-scale example solvers will be produced in order to determine the best method for implementing this change. More specifically, a problem will be set up and solved in a manner similar to how the deal.II wrapper classes function. This problem will store objects using Epetra, precondition with ML, and solve with AztecOO. This sample problem will then be updated to use the newer software packages, and time performance will be measured to determine the viability of the proposed method of updating deal.II.

Further, as the deal.II software relies upon numerical methods for solving partial differential equations, we want to explore the efficiency and accuracy of such methods. A one dimensional heat equation with Dirichlet boundary conditions and nonzero initial conditions will be solved analytically and using finite numerical methods to show the accuracy of finite numerical methods. This will include using the finite difference method and the finite element method. deal.II implements the finite element method, so the results obtained from using the finite element method will be analyzed in order to generally estimate the accuracy of deal.II.

4 Background

4.1 Analytical Solutions and Separation of Variables

Analytical solutions are obtained by using the traditional analytical methods that were developed by scientists to solve partial differential equations. Many different approaches exist for solving a PDE analytically that are chosen based on the specific problem being solved. In this thesis, based on the one dimensional heat equation chosen, the separation of variables method will be used to solve the heat equation. As the heat equation chosen is one dimensional and contains no sources, separation of variables can be used to easily solve our equation. Separation of variables is a method that can used to solve certain ordinary differential equations as well as linear, homogeneous PDEs. For a problem such as the one being solved in this thesis, the solution $u(x,t)$ is assumed to be of the form $u(x,t) = X(x)T(t)$. That is, it is assumed that the solution can be separated into two functions, each of one variable. This simplifies the problem as each equation $X(x)$ and $T(t)$ can be solved for separately.

4.2 The Finite Difference Method

The finite difference method of solving PDEs is a fairly simple numerical method. Without loss of generality, the method will be described assuming a one dimensional problem is being solved. First, the problem space is split into finitely many, evenly spaced points determined by a change in space variable, Δx. This discretization of the problem space is referred to as a mesh. The variable Δx is determined based on the desired number of points, M, and the total length of the problem space, l,

$$\Delta x = \frac{l}{M}.$$

Next, based on the desired number of time steps, N, a variable Δt is calculated by

$$\Delta t = \frac{T}{N}$$

where T is the last time at which a solution is desired [9]. The next steps in determining a solution are based on the scheme chosen to solve the problem.

There are several schemes in which the finite difference method can be used to solve a PDE. In this thesis, the forward in time central in space (FTCS) scheme was used [9]. Given a one-dimensional heat equation,

$$u_t - \alpha u_{xx} = 0,$$

the FTCS scheme can be used to find a solution at time $n+1$ of the form

$$u_{i,n+1} = (1-2\lambda)u_{i,n} + \lambda(u_{i+1,n} + u_{i-1,n})$$

where

$$\lambda = \frac{\alpha \Delta t}{(\Delta x)^2}.$$

In this case, $u_{i,n}$ refers to $u(i,n)$ where i is a point on the mesh, and n is a time step. In order to represent the entire problem space, the FTCS method can be expressed in the following matrix form

$$\begin{pmatrix} u_{0,n+1} \\ u_{1,n+1} \\ \vdots \\ u_{M-1,n+1} \end{pmatrix} = \begin{bmatrix} a & b & 0 & \cdots & 0 \\ b & a & b & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & b & a \end{bmatrix} \begin{pmatrix} u_{0,n} \\ u_{1,n} \\ \vdots \\ u_{M-1,n} \end{pmatrix}$$

where $a = 1 - 2\lambda$, $b = \lambda$, and n is a time step between 0 and $N - 1$ [9]. An approximate solution to the heat equation can then be found by refining the mesh until the values of $u_{i,n}$ are sufficiently close to a known analytical solution to the equation. One can refine the mesh by choosing increasingly smaller values for Δx. However, when this is done, it is important to note that consideration needs to be taken when subsequently choosing Δt in order to ensure that the solution remains stable [9]. Specifically, in order to maintain stability, λ has to be chosen so that $\lambda \leq \frac{1}{2}$, or more directly Δt must be chosen so that

$$\Delta t \leq \frac{\Delta x^2}{2\alpha}$$

[9]. With this condition, the Δt variable shrinks much more quickly than the Δx variable. Thus, with each refinement of the mesh, many more iterations are required to reach a solution at the same time step as in the previous refinement. This results in very large computation costs for implementations of the FTCS scheme that require a high level of accuracy. As a result, the FTCS method is not always a suitable method for solving partial differential equations.

4.3 The Finite Element Method

The finite element method (FEM) is another finite numerical method for solving partial differential equations. While the finite difference method is much easier to implement, the finite element method can handle more complex geometries and often times results in a more accurate approximation to the solution of a PDE. FEM takes a problem space and divides it into a finite number of nodes. This process is referred to as creating a mesh on the problem space. The nodes on the mesh are then used for creating an approximation to the solution of the partial differential equation.

Like the finite difference method, there are many different schemes or implementations of the finite element method. The implementation used in this thesis is the Crank-Nicolson scheme [7]. Unlike the FTCS scheme of the finite difference method, Crank-Nicolson requires that the solution u be solved for all nodes at any specific time as it is an implicit scheme. This can be seen in how the equation for solving the next time step is set up. Whereas the FTCS scheme can solve for any particular u_i at time t, the unknown or left-side of the equation used in the Crank-Nicolson scheme involves a linear combination of u_{i-1}, u_i, and u_{i+1}. As a result, the Crank-Nicolson scheme tends to be quite a bit more accurate than the FTCS scheme of finite difference. Further, unlike the FTCS scheme, the Crank-Nicolson scheme is unconditionally stable. That is, unlike FTCS in which the Δt has to be carefully chosen in order for the scheme to work properly, there is no concern over appropriate choice of Δt in Crank-Nicolson [7].

4.4 Trilinos Software

The Trilinos Project is a large collection of packages designed to solve large scale, complex, physics, engineering, and science problems. This involves having the capability to solve linear and non-linear systems of equations, eigensystems, and other similar problems. The packages that make up Trilinos are self-contained, independent pieces of software with their own requirements. However, Trilinos allows for and provides various ways for packages to interact with one another. Further, Trilinos allows for integration of Trilinos with other established libraries of solvers. The Trilinos Project rose out of a desire and perceived need to provide a common algorithmic base for solving large-scale scientific problems. As a whole, Trilinos acts as a very valuable starting point for developing new algorithms and enabling technologies. A core goal of the Trilinos Project is to develop parallel solvers to enable larger problems to be solved using numerical methods. As a result, Trilinos is designed to allow for use on distributed parallel architectures. While Trilinos contains many packages, the ones used in this thesis are Epetra, Tpetra, Xpetra, ML, MueLu, AztecOO, and Belos [1].

4.4.1 Epetra, Tpetra, and Xpetra

Epetra, Tpetra, and Xpetra are all packages that allow for the storage and creation of objects such as vectors, matrices, and graphs. Petra is Greek for foundation, and as such, these packages provide the basis for all solvers in Trilinos. The ”e” in Epetra stands for essential and is the C++ production implementation of the Petra model. Epetra supports real, double-precision, floating point data. Epetra also offers an Epetra-64 extension which allows for the use of 64-bit indices. Further, it provides for serial, parallel, and distributed memory capabilities. Epetra avoids using more advanced C++ capabilities such as templates in order to maintain high levels of portability and stability. Epetra is highly useful for developing new solvers and using existing solvers as it handles the intricacies of parallel execution.

Tpetra on the other hand is an implementation of the Petra model that allows for templated C++. It implements scalar and ordinal fields as template types. As a result, Tpetra supports many more data types than Epetra; in principal, a scalar can be implemented as any abstract data type so long as the data type supports basic mathematical operations. Further, the ordinal type can be any abstract data type that supports some form of indexing. Like Epetra, Tpetra is designed for a general parallel distributed memory machine and as a result supports parallelism in execution.

Xpetra is in general, a lightweight wrapper to both Tpetra and Epetra. In this way, a programmer can write one object and be able to specify later whether to use Epetra or Tpetra. The syntax of Xpetra models that of Tpetra as it also allows for templates. Xpetra is the package used by MueLu [1].

4.4.2 ML and MueLu

Both ML and MueLu are multigrid preconditioner packages within Trilinos. Preconditioners are methods which approximate the inverse of a matrix in a linear system. This then helps iterative solvers to solve the linear system using fewer iterations. ML is designed to handle large sparse linear systems typically derived from elliptic partial differential equation discretizations. In addition to providing the option of building and designing multigrid preconditioners and solvers, ML also contains several black-box classes to allow for scalable smoothed aggregation preconditioners. ML is currently the main multigrid preconditioner package of Trilinos.

MueLu is also a package designed to precondition large, sparse linear systems of equations. It is designed to be a flexible, high performance, multigrid solver library. MueLu is also meant to be easy to use and provide support for many different platforms ranging from personal computers to large, parallel clusters [1].

4.4.3 AztecOO and Belos

In Trilinos, AztecOO and Belos are both solver packages. AztecOO is an object-oriented interface of the Aztec solver. It supports flexible construction of matrix and vector arguments through the use of Epetra objects. AztecOO produces iterative solutions of large sparse linear systems.

Belos is described as providing "next-generation" iterative linear solvers. It also serves as a powerful linear solver developer framework. Belos treats matrices, vectors, and preconditioners as black-box objects allowing for any combination of matrix, preconditioner, and vector types that reasonably make sense together. Belos is designed to work efficiently and provide flexibility [1].

4.4.4 Summary of Trilinos Packages

Package Name	Package Description	Use in Thesis
Epetra	Essential petra software package. Supports double-precision, floating point data. Storage and creation of objects such as matrices and vectors	Currently used in deal.II wrapper classes to store objects.
Tpetra	Templated petra software. Supports scalar and ordinal data types. Creates and stores objects such as matrices and vectors.	Not explicitly used in this thesis. Often is available as an option when Xpetra is used.
Xpetra	Acts as a wrapper on top of Epetra and Tpetra.	Used with MueLu in proposed method of updating deal.II. Worked as wrapper on top of Epetra objects.
ML	Multigrid preconditioner package. Main multigrid preconditioner package in Trilinos	Currently used as the preconditioner in deal.II wrapper classes.
MueLu	Newer multigrid preconditioner software. Flexible, high-performance library.	Used as a preconditioner in proposed method of updating deal.II.
AztecOO	Iterative solver. Object oriented version of Aztec software. Supports flexible construction of arguments	Currently used as solver in deal.II wrapper classes.
Belos	Next generation iterative solver. Powerful linear solver developer framework	Used as a solver in proposed method of updating deal.II. [1]

4.5 deal.II Software

deal.II, the successor to the Differential Equations Analysis library, is a C++ program library for the solving of partial differential equations using finite elements. deal.II allows for rapid development of finite element codes by offering interfaces for adaptive meshes and other objects and tools typically used in finite element programs. It is used in both academic and commercial products. deal.II utilizes templates so that the programmer can essentially write code independent of the dimensions of the problem being solved; that is, the code can be written independent of whether the user is solving a 1D, 2D, or 3D problem. Currently, the deal.II software is supported on Linux and Mac OS X. Due to the fact that deal.II solvers do not currently implement any form of parallelism, a user with the Trilinos software can link Trilinos to deal.II when installing. deal.II provides wrapper classes that provide an almost identical interface to deal.II's own linear solvers. By utilizing Trilinos when creating a deal.II program, the user is able to take advantage of the parallel capabilities of Trilinos specifically parallelism based on MPI.

5 Methods

5.1 Source Code and Computing Architecture

For analyzing the summations and plotting the results of the analytical solution, *Mathematica*, a technical software was used. Mathematica was also used to process the matrix-vector multiplication and plotting of results for the finite difference method. Matlab, a language for technical computing, was used to program the Crank-Nicolson scheme of the finite element method and to plot the results. The Trilinos software packages including MueLu, ML, Belos, AztecOO, Xpetra, and Epetra were used for the small scale examples implementing the proposed changes to deal.II. The version of the Trilinos software that was used was downloaded from the Trilinos public repository in November, 2014. These examples were programmed in the C++ programming language.

A personal laptop and basic CSBSJU windows client machine were used in conjunction with Matlab and Mathematica software to solve the one dimensional heat equation. An eight node cluster, named Melchior, was used for running the time trials on the small scale examples. Every node of Melchior (numbered 0 through 7) was used for these time trials. Each node of Melchior is equipped with an Intel Xeon processor.

5.2 Solving the One Dimensional Heat Equation Analytically

For this thesis, a one dimensional heat equation with zero Dirichlet boundary conditions and nonzero initial conditions was selected. The problem can be described as follows:

$$u_t(x,t) = \frac{1}{10}u_{xx}(x,t)$$

$$0 < x < 6$$

$$t > 0$$

$$u(x,0) = 6x - x^2$$

$$u(0,t) = 0 = u(6,t)$$

In order to solve the problem analytically, the separation of variables method was used. As previously mentioned, this means assuming that our function $u(x,t)$ can be written in terms of two, single variable functions. That is, $u(x,t) = X(x)T(t)$. Substituting this into our problem, we obtain the following new problem:

$$X(x)T'(t) = \frac{1}{10}X''(x)T(t).$$

Isolating each variable to one side of the equality, the problem becomes

$$\frac{10T'(t)}{T(t)} = \frac{X''(x)}{X(x)}.$$

Because the left side of the equality is a function of t and the right a function of x, it can be said that both sides are constant. That is we can say each side is equal to some constant λ. Using this fact, two ordinary differential equations (ODEs) can be obtained. They are

$$X'' - \lambda X = 0$$

and

$$T' - \frac{1}{10}\lambda T = 0.$$

As X ranges from 0 to 6 in our problem and homogeneous Dirichlet boundary conditions $u(0,t) = 0 = u(6,t)$ were chosen, it becomes clear that for the ODE $X'' - \lambda X = 0$, $\lambda < 0$. If $\lambda \geq 0$, the only solutions to $X'' - \lambda X = 0$ would be identically zero. Letting $\lambda = \alpha^2$, the general solution of the ODE $X'' - \alpha^2 X = 0$ becomes

$$X(x) = c_1 \cos \alpha x + c_2 \sin \alpha x.$$

To satisfy the boundary conditions, we assume $X(0) = X(6) = 0$. By looking at

$$X(0) = c_1 \cos \alpha 0 + c_2 \sin \alpha 0 = c_1$$

it can be seen that $c_1 = 0$. As we do not want $X(x) = 0$, this means that c_2 cannot be 0. Thus, because $X(6) = 0$ and $c_2 \neq 0$, $\sin \alpha 6 = 0$. That is, $6\alpha = n\pi$ for some n. The solution for X

becomes

$$X(x) = c_2 \sin \frac{n\pi x}{6}, n = 1, 2, 3, \ldots.$$

At this point, we consider the ODE

$$T' - \frac{1}{10}\alpha^2 T = 0.$$

As we have solved for α, this can be substituted into the equation to obtain

$$T' - \frac{n^2\pi^2}{10 * 36} T = 0, n = 1, 2, 3, \ldots.$$

The general solution to this ODE is

$$T = ce^{\frac{-n^2\pi^2 t}{10*36}}, n = 1, 2, 3, \ldots.$$

Combining the two general solutions, we obtain the following form of a solution to our PDE

$$u(x,t) = b_n e^{\frac{-n^2\pi^2 t}{10*36}} \sin \frac{n\pi x}{6}, n = 1, 2, 3, \ldots.$$

However, as the initial conditions of our problem are not trigonometric, the form of the solutions becomes

$$u(x,t) = \sum_{n=1}^{\infty} b_n e^{\frac{-n^2\pi^2 t}{360}} \sin \frac{n\pi x}{6}$$

where the initial conditions become

$$6x - x^2 = \sum_{n=1}^{\infty} b_n \sin \frac{n\pi x}{6}.$$

In order to solve for the coefficients b_n, both sides of the equation are multiplied by $\sin \frac{n\pi x}{6}$ and are integrated from 0 to 6. The problem then becomes

$$\int_0^6 (6x - x^2) \sin \frac{n\pi x}{6} dx = \sum_{n=1}^{\infty} b_n \int_0^6 \sin^2 \frac{n\pi x}{6} dx$$

By utilizing the Mathematica software to aid in integration, we solved for

$$b_n = \frac{72(2 - 2(-1)^n)}{n^3\pi^3}.$$

Noting that for any even n the coefficient $b_n = 0$, this became

$$b_n = \frac{288}{(2n+1)^3\pi^3}.$$

Having solved for the constants b_n, the solution obtained for our one dimensional heat equation is

$$u(x,t) = \sum_{n=1}^{\infty} \frac{288}{(2n+1)^3\pi^3} e^{\frac{-n^2\pi^2 t}{360}} \sin\frac{n\pi x}{6}.$$

At this point, the function was graphed in Mathematica at various times and over the time interval $[0, 25]$ [4].

5.3 Solving Using FTCS Finite Differences

As described earlier, the FTCS scheme for finite difference was chosen as one numerical method for solving the one dimensional heat equation. Initially the variables were chosen as follows: $\Delta x = 1$, $\Delta t = 1$. Each subsequent iteration of the method involved halving the change in x variable, Δx. The change in t variable, Δt was then chosen so that the solution would remain stable by considering the requirement that $\lambda \leq \frac{1}{2}$. As mentioned, Mathematica was used for computing the matrix-vector products and displaying results.

Example: Solving the 1D Heat Equation using FTCS with $\mathbf{\Delta x = 1}$, $\mathbf{\Delta t = 1}$

With $\Delta x = 1$, $M = 6$. For each $x_i = i\Delta x$, $i = 0, 1, \ldots, M$. With $\Delta t = 1$, $N = 10$, and for each $t_i = i\Delta t$, $i = 0, 1, \ldots, N$. This choice of M results in a 6 x 6 matrix and vectors composed of 6 elements. To ensure the system will be stable, it can be seen that the condition on λ is met as

$$\lambda = \frac{1}{10} * \frac{1}{1} = 0.1 \leq \frac{1}{2}.$$

At this point, the values of the initial vector can be determined. To do this, each element is computed as

$$u_{0,i} = 6x_i - x_i^2.$$

The resulting initial vector is then $\begin{pmatrix} 0, 5, 8, 9, 8, 5, 0 \end{pmatrix}^T$.

To assemble the tridiagonal matrix used in the FTCS scheme a and b must be computed. Note for this problem, $\alpha = \frac{1}{10}$. We can then compute

$$\lambda = \frac{\alpha\Delta t}{\Delta x^2} = \frac{\frac{1}{10}1}{1^2} = \frac{1}{10} = .1.$$

Next, a and b can be computed as follows:

$$a = 1 - 2\lambda = 1 - 2(.1) = .8$$

$$b = \lambda = .1$$

The tridiagonal matrix used for this example is then $\begin{bmatrix} .8 & .1 & 0 & 0 & 0 & 0 \\ .1 & .8 & .1 & 0 & 0 & 0 \\ 0 & .1 & .8 & .1 & 0 & 0 \\ 0 & 0 & .1 & .8 & .1 & 0 \\ 0 & 0 & 0 & .1 & .8 & .1 \\ 0 & 0 & 0 & 0 & .1 & .8 \end{bmatrix}$ As $N = 10$, to solve for $u(x, 10)$, the following matrix vector multiplication is conducted:

$$\begin{pmatrix} u_{0,10} \\ u_{1,10} \\ u_{2,10} \\ u_{3,10} \\ u_{4,10} \\ u_{5,10} \\ u_{6,10} \end{pmatrix} = \begin{bmatrix} .8 & .1 & 0 & 0 & 0 & 0 \\ .1 & .8 & .1 & 0 & 0 & 0 \\ 0 & .1 & .8 & .1 & 0 & 0 \\ 0 & 0 & .1 & .8 & .1 & 0 \\ 0 & 0 & 0 & .1 & .8 & .1 \\ 0 & 0 & 0 & 0 & .1 & .8 \end{bmatrix}^{10} \begin{pmatrix} 0 \\ 5 \\ 8 \\ 9 \\ 8 \\ 5 \\ 0 \end{pmatrix}$$

When computed this results in the solution vector

$$u(x,t) = \Big(1.99502, 4.32822, 6.33233, 7.11653, 6.33233, 4.32822, 1.99502\Big)^T.$$

5.4 Solving Using Crank-Nicolson FEM

In order to solve the one dimensional heat equation using the Crank-Nicolson scheme of finite element computations, a Matlab document was modified to fit our specific problem [6]. The file used two other Matlab files that were obtained from the same source. One file completed LU-factorization of a tridiagonal matrices and the other solved linear systems with LU-factored matrices of coefficients. The exact approximation used by the Crank-Nicolson scheme is

$$-\frac{\alpha}{2\Delta x^2}u_{i-1}^{k+1} + (\frac{1}{\Delta t} + \frac{\alpha}{2\Delta x^2})u_i^{k+1} - \frac{\alpha}{2\Delta x^2}u_{i+1}^{k+1} = \frac{\alpha}{2\Delta x^2}u_{i-1}^{k} + (\frac{1}{\Delta t} - \frac{\alpha}{2\Delta x^2})u_i^{k} + \frac{\alpha}{2\Delta x^2}u_{i+1}^{k}$$

where for our heat equation, $\alpha = \frac{1}{10}$, all values on the left side of the equation are unknown, and values on the right side are known. This equation is converted into the following linear system

$$\begin{bmatrix} a & b & 0 & \cdots & 0 \\ c & a & b & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & c & a \end{bmatrix} \begin{pmatrix} u_1^{k+1} \\ u_2^{k+1} \\ \vdots \\ u_N^{k+1} \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_N \end{pmatrix}$$

where

$$a = \frac{1}{\Delta t} + \frac{\alpha}{\Delta x^2}$$

$$b = c = -\frac{\alpha}{2\Delta x^2}$$

$$d_i = -cu_{i-1}^k + (\frac{1}{\Delta t} + b + c)u_i^k - bu_{i+1}^k$$

[7]. The Matlab program then solves this system of equations using LU-factorization. The system is iterated multiple times until the desired time step is reached. This program was run similarly to the finite difference in that the same values for Δx and Δt were used. Further, the point $u(3, 10)$ was recorded for each discretization.

5.5 Developing a Method for Updating the deal.II Software

After developing an understanding of finite methods for solving partial differential equations, a method for updating the deal.II software was developed. To begin, an example using Epetra objects, and ML preconditioner, and an AztecOO solver was taken from the Trilinos examples directory. The example generated a three dimensional Laplacian matrix and solved a test linear problem obtained from the gallery object. The problem size could be changed to any value specified (line 61 of MLAztecOO.cpp in Appendix) so long as the value was a perfect cube (as the problem was dealing with a three dimensional matrix). The next step was to modify the program so that a Belos solver was used instead of an AztecOO solver. This was a relatively simple replacement using the Trilinos documentation for Belos available online as a guide. The next step was to determine how to translate Epetra objects into Xpetra objects. Xpetra had a method to accomplish this for vector objects. The vectors simply had to be wrapped as reference-counted pointers (RCPs) before being passed into the method (lines 115-122 of MueLuBelos.cpp in Appendix). However, the translation of the matrix was a bit more involved. First, the graph of the matrix (containing the information describing locations of nonzero values) had to be copied into an empty Xpetra object. Then, the values had to be copied one row at a time into the Xpetra object (lines 100-111 of MueLuBelos.cpp in Appendix). Once this was accomplished, a MueLu preconditioner and Belos solver could be set up to solve the linear problem. After the three examples were working correctly, a timer was added to output the total runtime and solving time for each example. The examples were named MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp.

In order to test these examples, the following time sizes were selected: 50^3, 120^3, 190^3, 260^3, 330^3, and 400^3. These were selected based off of initial time trials to determine a range of problem sizes that would obtain a wide range of total time results. The examples were compiled using a basic Makefile obtained with the MLAztecOO.cpp file from the Trilinos examples directory. On each problem size, each example was run 10 times on a node of Melchior. Two scatter plots, total time and solve time plots, were then created on each problem size from these results.

6 Results

6.1 One-Dimensional Heat Equation Results

6.1.1 Analytical Solution

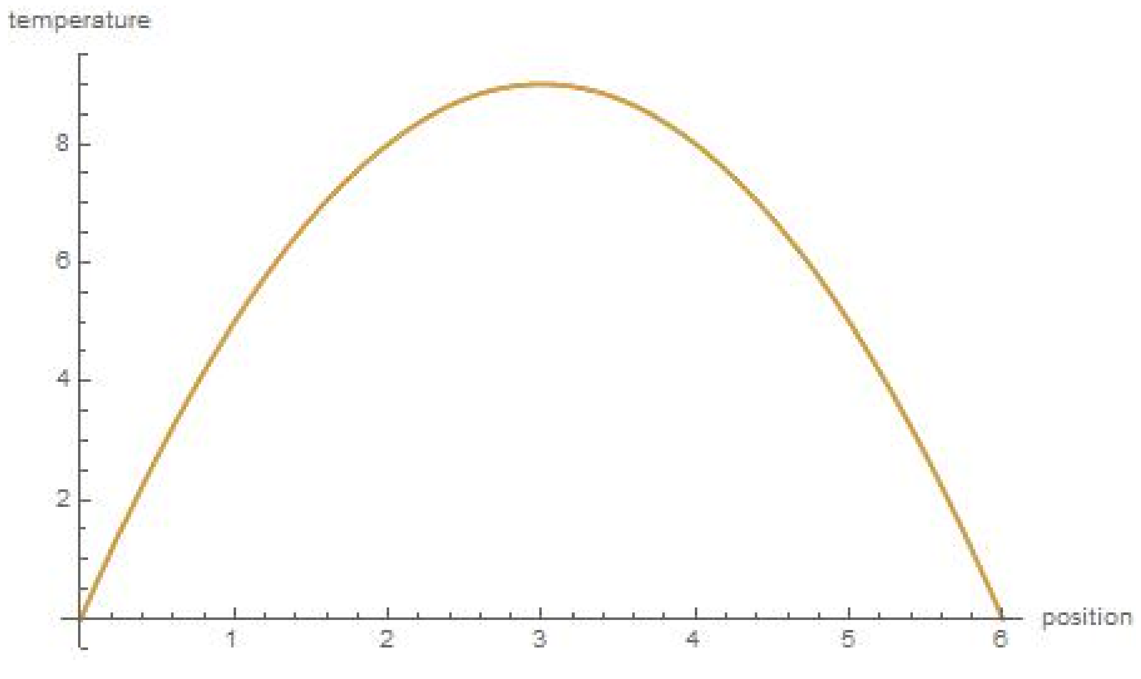

Figure 1: An overlay of the graph of the computed analytical solution at time, $t = 0$ and the graph of the specified initial conditions, $u(x, 0) = 6x - x^2$.

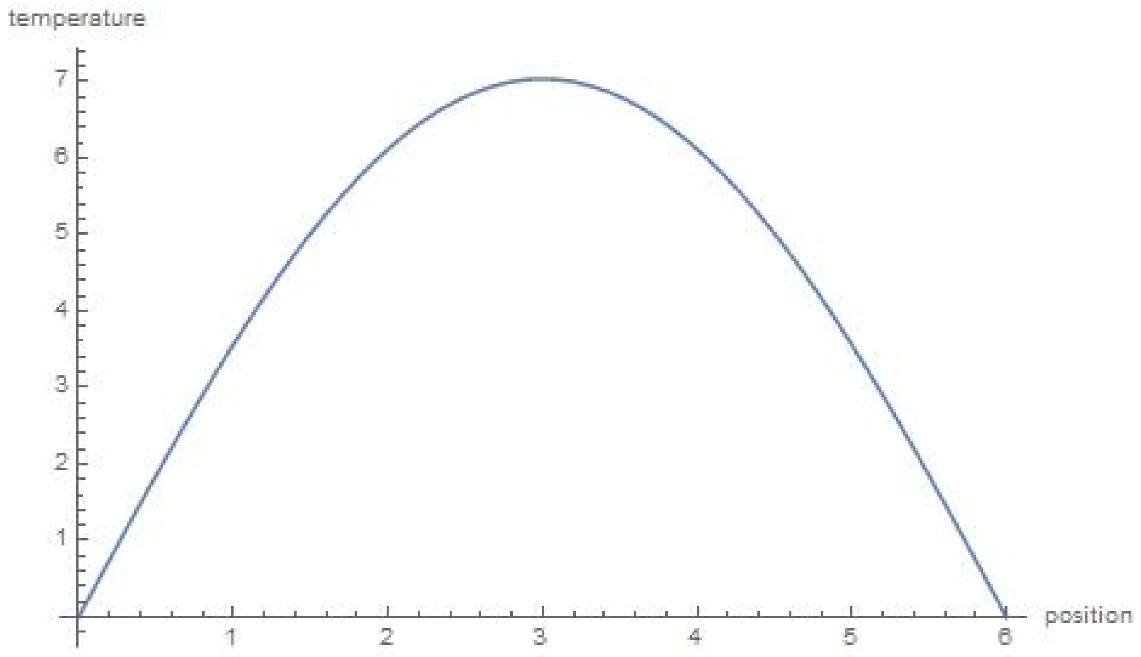

Figure 2: The graph of the computed analytical solution at time, $t = 10$.

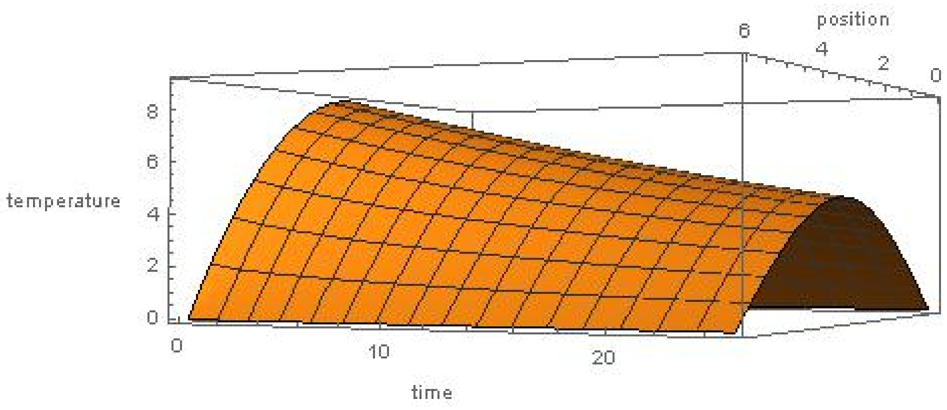

Figure 3: The graph of the computed analytical solution over the period of time $t = 0$ to $t = 25$.

By analyzing Figure 1, it can be seen that the analytical solution to the one dimensional heat equation obtained using separation of variables,

$$u(x,t) = \sum_{n=0}^{\infty} \frac{288}{(2n+1)^3\pi^3} e^{\frac{-(2n+1)^2\pi^2 t}{360}} \sin(\frac{(2n+1)\pi x}{6})$$

is correct at time, $t = 0$. Further, by looking at both Figure 2 and Figure 3, it can be seen that the function is behaving as expected. That is, as there is not heat source or sink, it is expected that the heat along the rod would dissipate over time.

6.1.2 Finite Difference Results

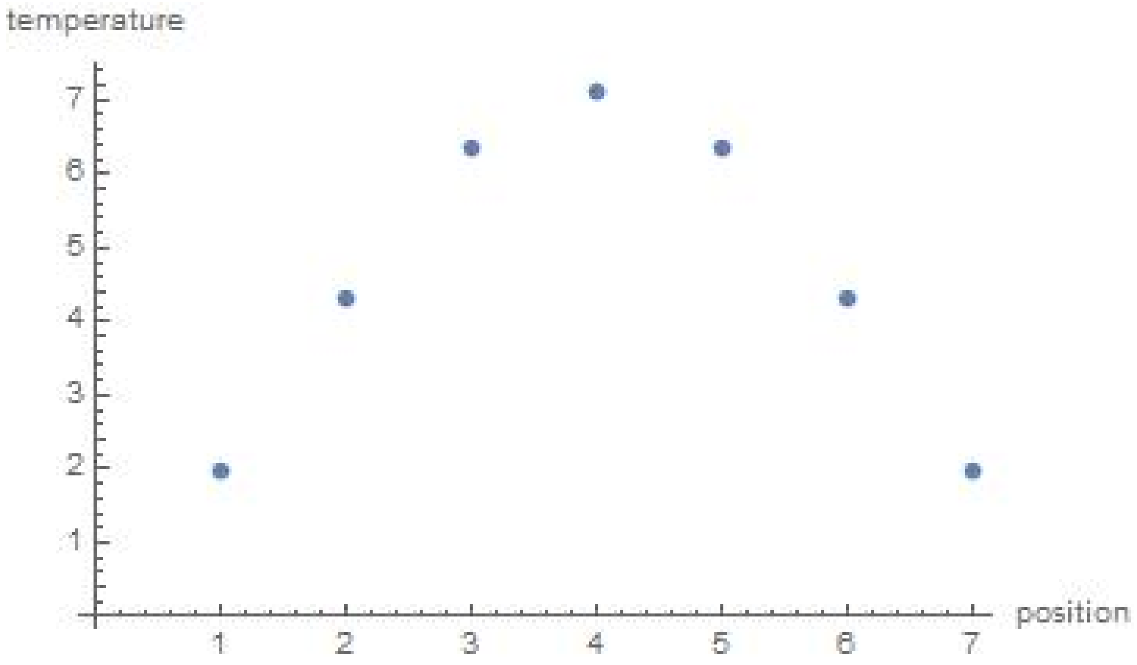

Figure 4: The graph of the solution computed using the FTCS scheme with $\Delta x = 1$ and $\Delta t = 1$ at time $t = 10$.

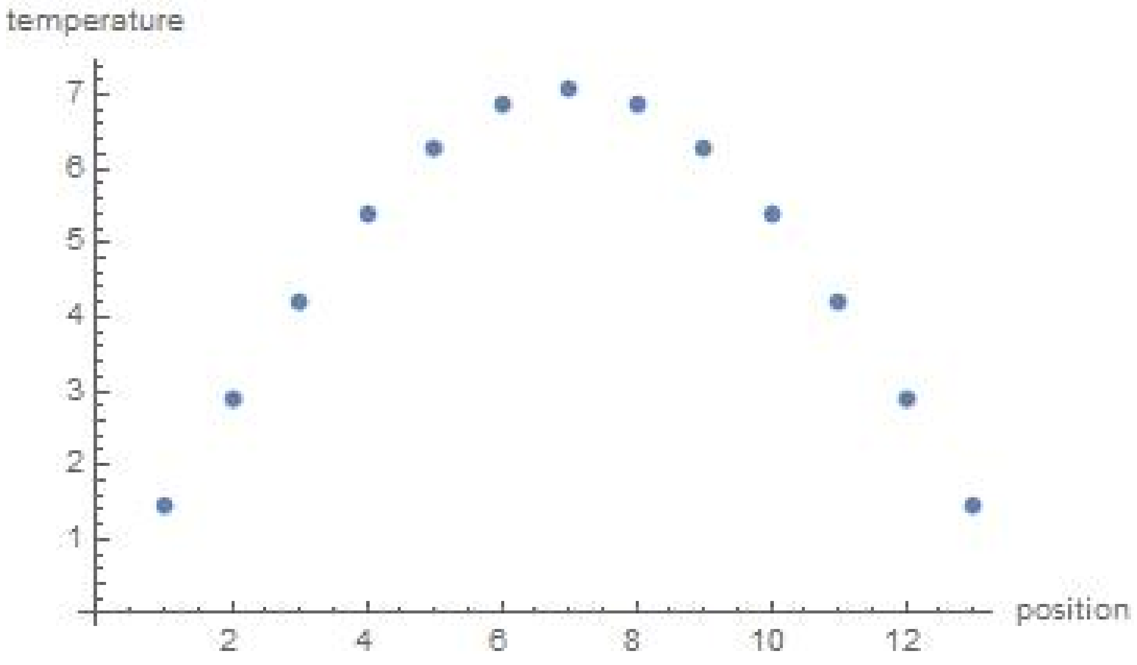

Figure 5: The graph of the solution computed using the FTCS scheme with $\Delta x = \frac{1}{2}$ and $\Delta t = 1$ at time $t = 10$.

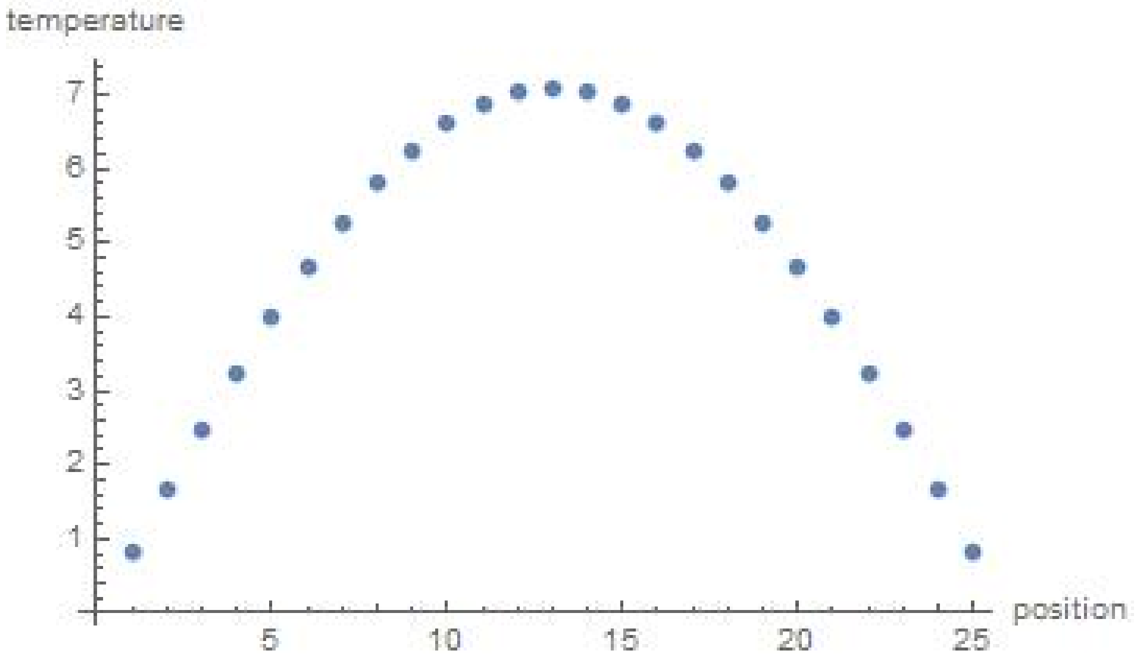

Figure 6: The graph of the solution computed using the FTCS scheme with $\Delta x = \frac{1}{4}$ and $\Delta t = \frac{1}{4}$ at time $t = 10$.

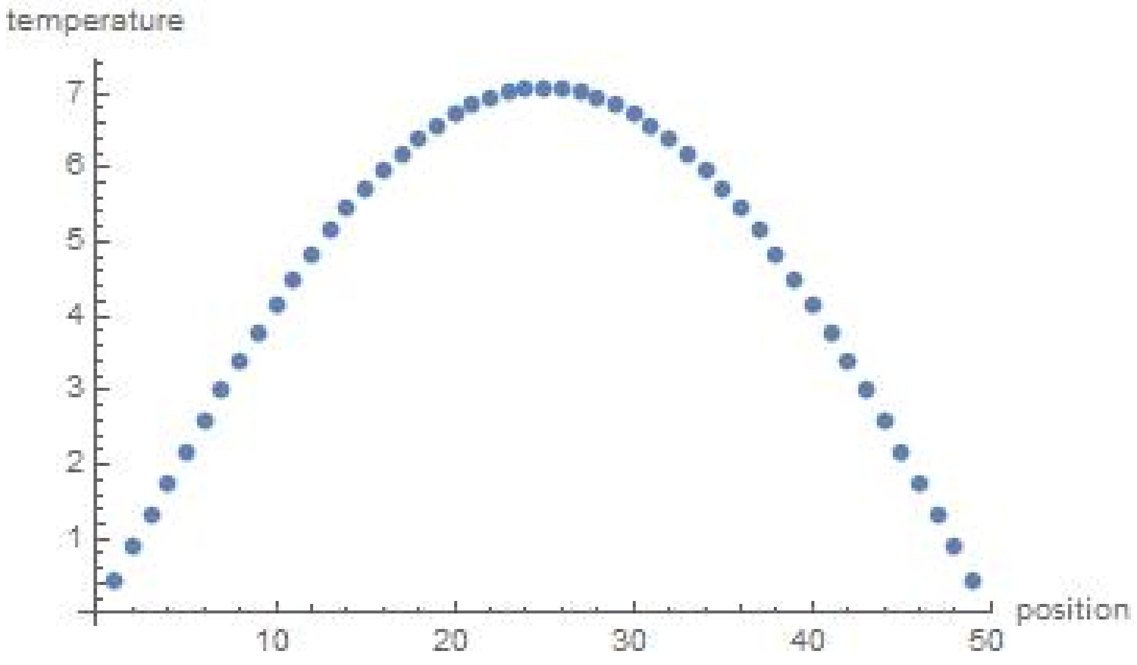

Figure 7: The graph of the solution computed using the FTCS scheme with $\Delta x = \frac{1}{8}$ and $\Delta t = \frac{1}{16}$ at time $t = 10$.

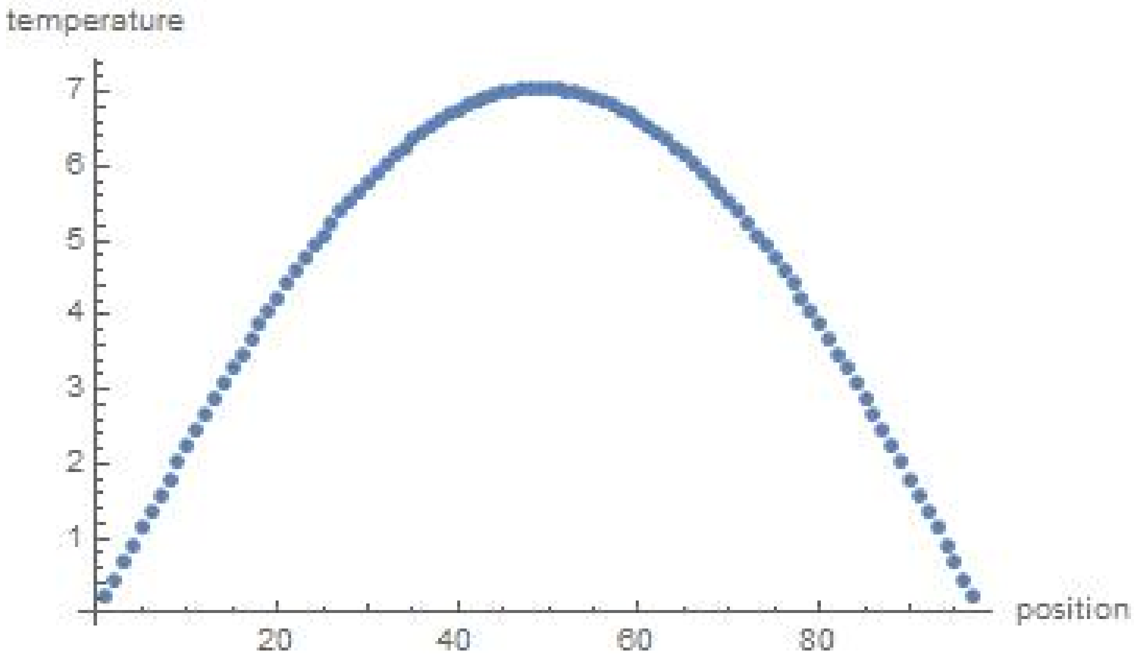

Figure 8: The graph of the solution computed using the FTCS scheme with $\Delta x = \frac{1}{16}$ and $\Delta t = \frac{1}{64}$ at time $t = 10$.

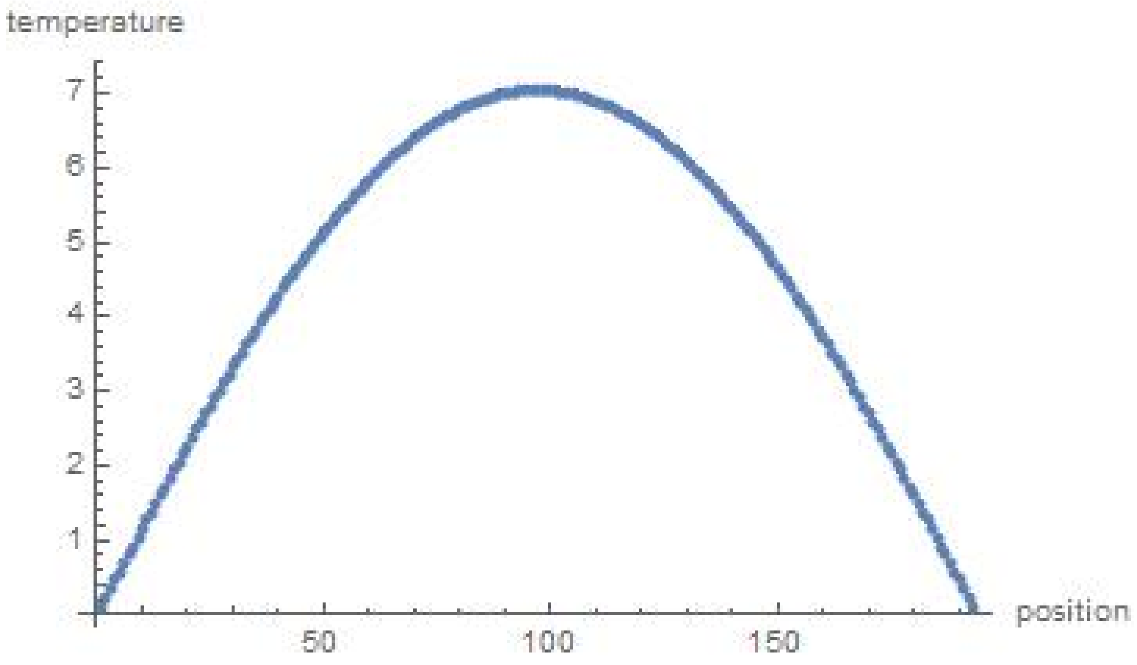

Figure 9: The graph of the solution computed using the FTCS scheme with $\Delta x = \frac{1}{32}$ and $\Delta t = \frac{1}{256}$ at time $t = 10$.

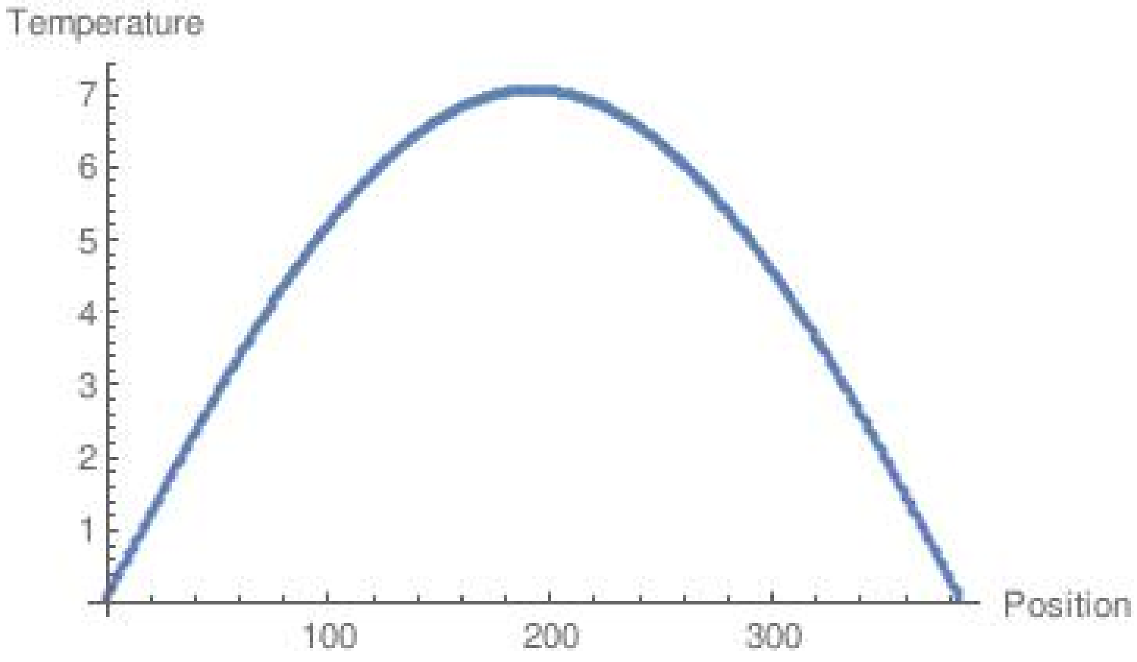

Figure 10: The graph of the solution computed using the FTCS scheme with $\Delta x = \frac{1}{64}$ and $\Delta t = \frac{1}{1024}$ at time $t = 10$.

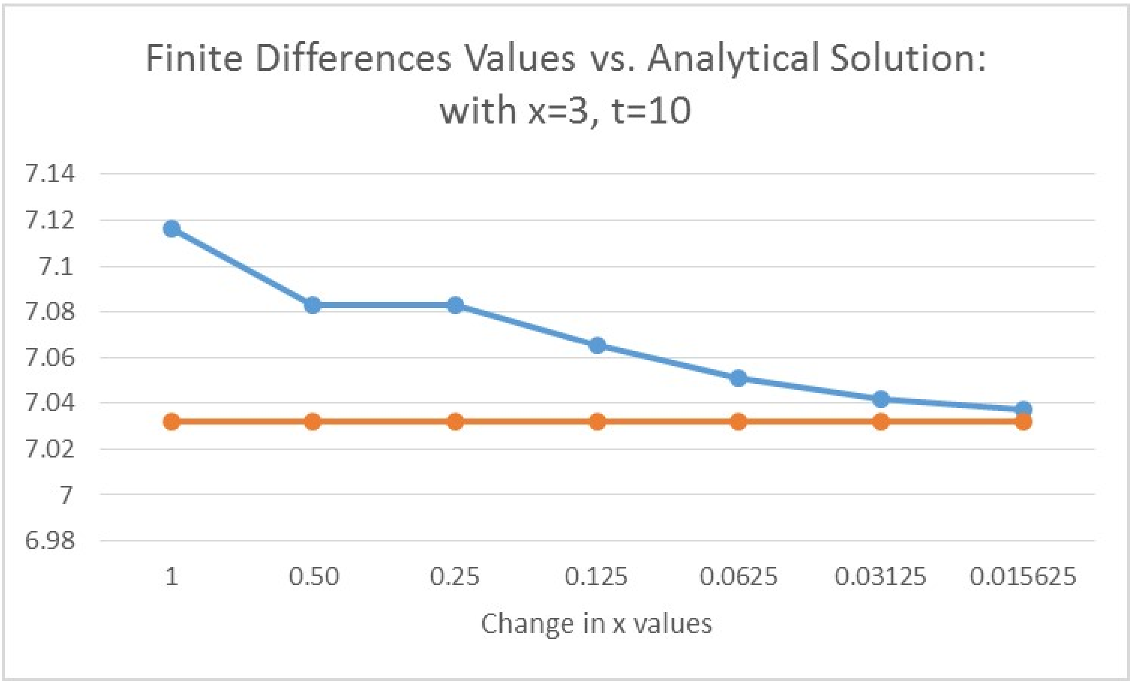

Figure 11: The graph compares the values of the analytical solution and the values of the solution obtained using the finite difference method at $x = 3$ and $t = 10$, i.e. $u(3, 10)$. The constant line is the value obtained from the analytical solution and the line decreasing in value corresponds to the values obtained from the finite difference method for decreasing Δx.

Table 1: Results of Finite Difference

Δx	Δt	$u(3,10)$
1	1	7.11653
$\frac{1}{2}$	1	7.08331
$\frac{1}{4}$	$\frac{1}{4}$	7.08334
$\frac{1}{8}$	$\frac{1}{16}$	7.06539
$\frac{1}{16}$	$\frac{1}{64}$	7.05089
$\frac{1}{32}$	$\frac{1}{256}$	7.04207
$\frac{1}{64}$	$\frac{1}{1024}$	7.03723
Analytical	-	7.03211

Having verified the analytical solution to the one dimensional heat equation, the accuracy of the FTCS scheme of the finite difference method can be analyzed. Figures 4-10 show that as both the number of time steps and the number of points on the problem space increase, the approximation of the method converges to the analytical solution. It should be noted that this method does not maintain zero temperature at the endpoints. However, as the problem is further discretized, the end point values do tend back towards zero. Figure 11 shows this convergence by plotting the values for $u(3, 10)$ obtained from FTCS scheme against the value obtained from the analytical solution. While the approximation achieved at $\Delta x = \frac{1}{64}$ and $\Delta t = \frac{1}{1024}$ is very close to the analytical solution (roughly .05 off), it was not possible to further refine the problem and get a better approximation without causing the Mathematica software to crash. Thus, while the FTCS scheme of the finite difference method is simple and easy to implement, it is not a realistic method when either a large problem is being solved or when a very close approximation is required.

6.1.3 Finite Element Method Results

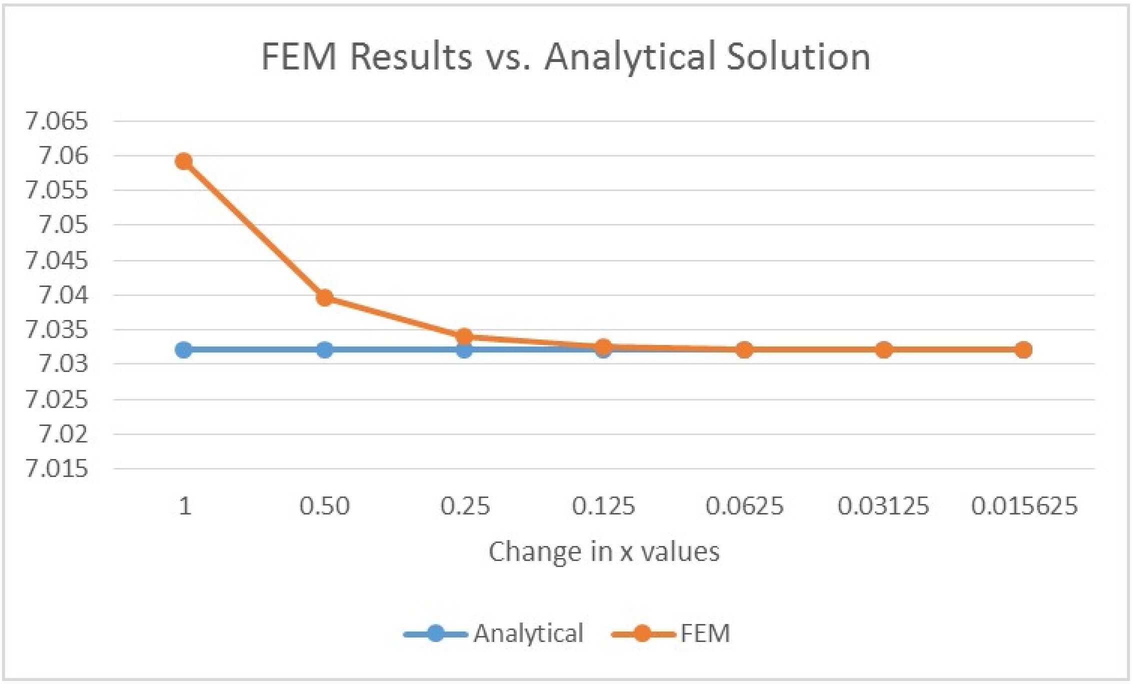

Figure 12: The graph compares the values of the analytical solution and the values of the solution obtained using the Crank Nicolson finite element method at $x = 3$ and $t = 10$, i.e. $u(3, 10)$. The solutions are obtained from the Matlab implementation of this method. The constant line is the value obtained from the analytical solution and the line decreasing in value corresponds to the values obtained from the finite element method for decreasing Δx.

Table 2: Results of Finite Element Method

Δx	Δt	$u(3,10)$
1	1	7.05911
$\frac{1}{2}$	1	7.0396
$\frac{1}{4}$	$\frac{1}{4}$	7.0340
$\frac{1}{8}$	$\frac{1}{16}$	7.0326
$\frac{1}{16}$	$\frac{1}{64}$	7.0322
$\frac{1}{32}$	$\frac{1}{256}$	7.0321
$\frac{1}{64}$	$\frac{1}{1024}$	7.0321
Analytical	-	7.03211

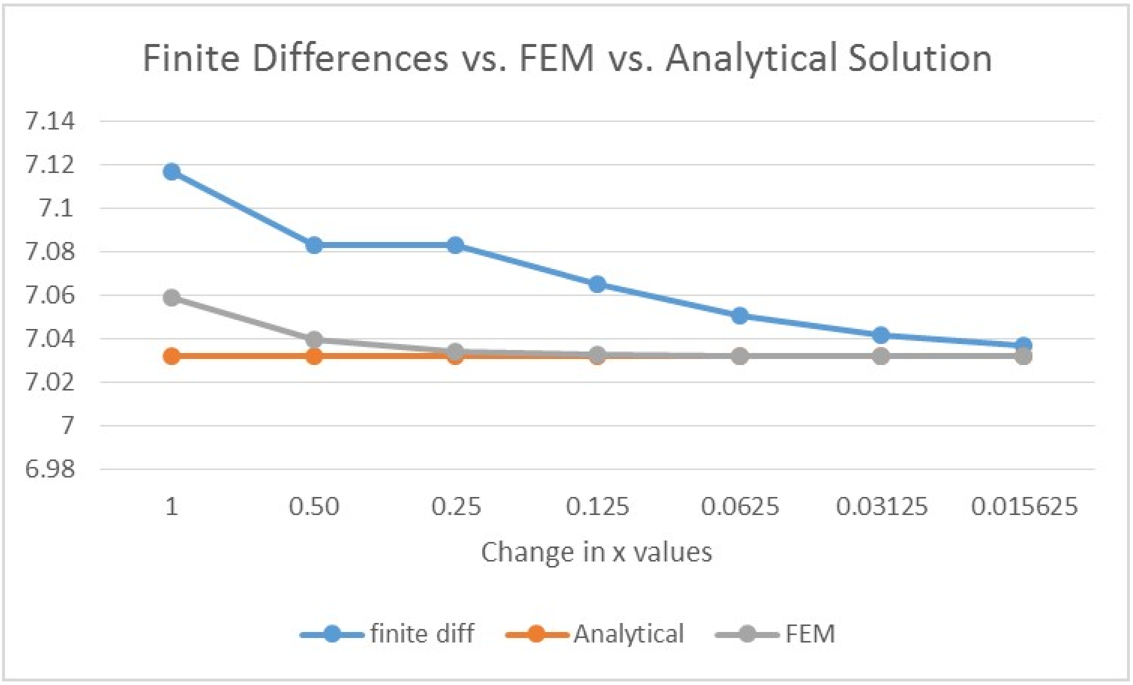

Figure 13: The graph compares the value of the analytical solution, the values of the solution obtained using the Crank Nicolson finite element method, and the values of the solution obtained using the FTCS finite difference scheme at $x = 3$ and $t = 10$, i.e. $u(3, 10)$. The solutions from the finite element method are obtained from the Matlab implementation of this method. The constant line is the value obtained from the analytical solution and the line decreasing in value corresponds to the values obtained from the finite element method for decreasing Δx.

When looking at the results of the Crank-Nicolson scheme of the finite element method, specifically Figure 13, it becomes clear that this method provides a much better approximation than the finite difference approach. It converged to the solution much more quickly than the finite difference method did. A coarser discretization of the problem was able to produce much better results for $u(3, 10)$ than the same discretization using FTCS. As a result, the Crank-Nicolson scheme of the finite element method appears to be an adequate choice for problems where close approximations are required.

6.2 Small Scale Example Results

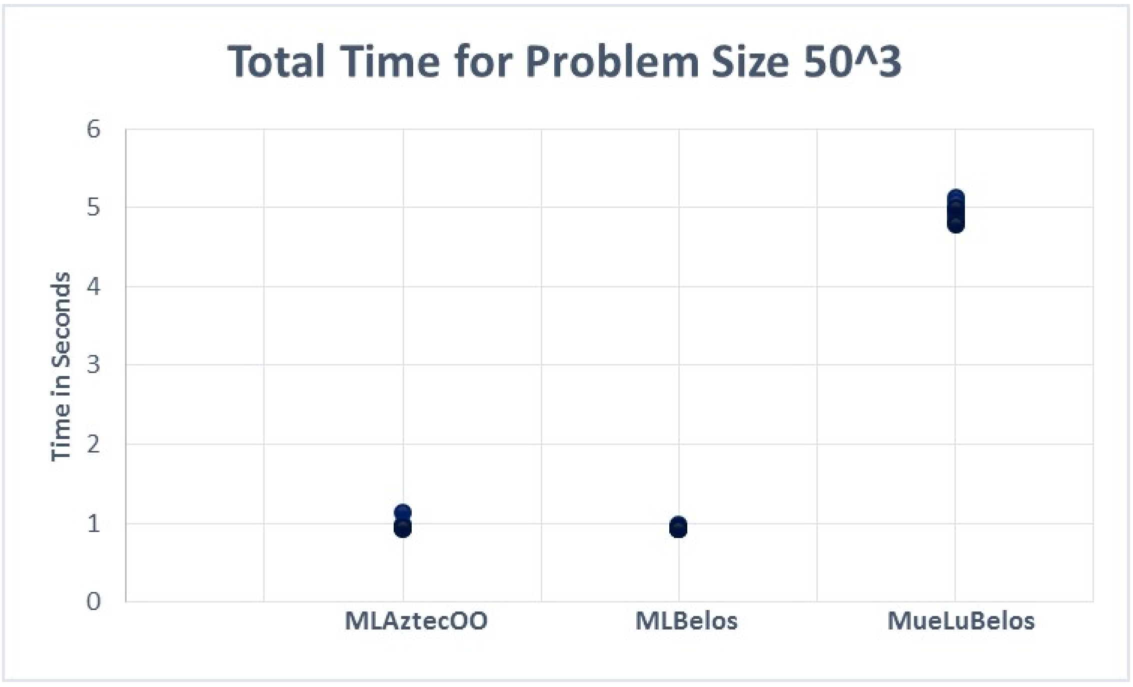

Figure 14: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 50^3. The measured time is total time to run including problem setup and solving. This plot shows the range in time performance for each file and the comparative time to complete between each file.

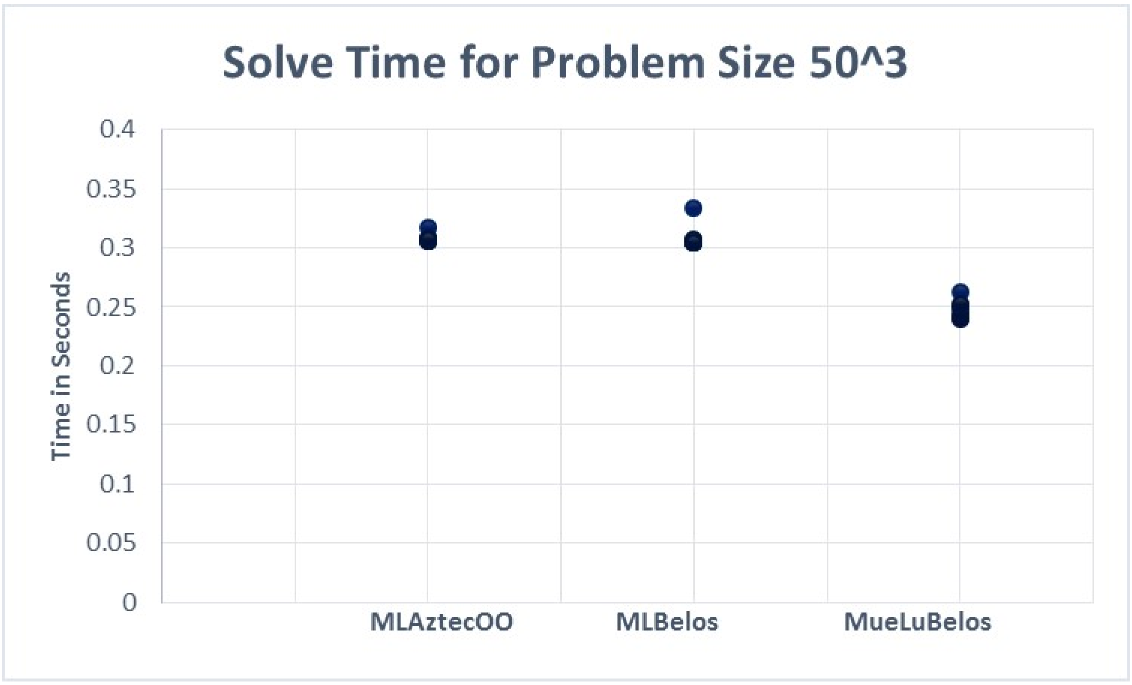

Figure 15: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 50^3. The measured time is the amount of time taken to solve the problem. Time to setup is not included. This plot shows the range in solving time performance for each file and the comparative time to solve between each file.

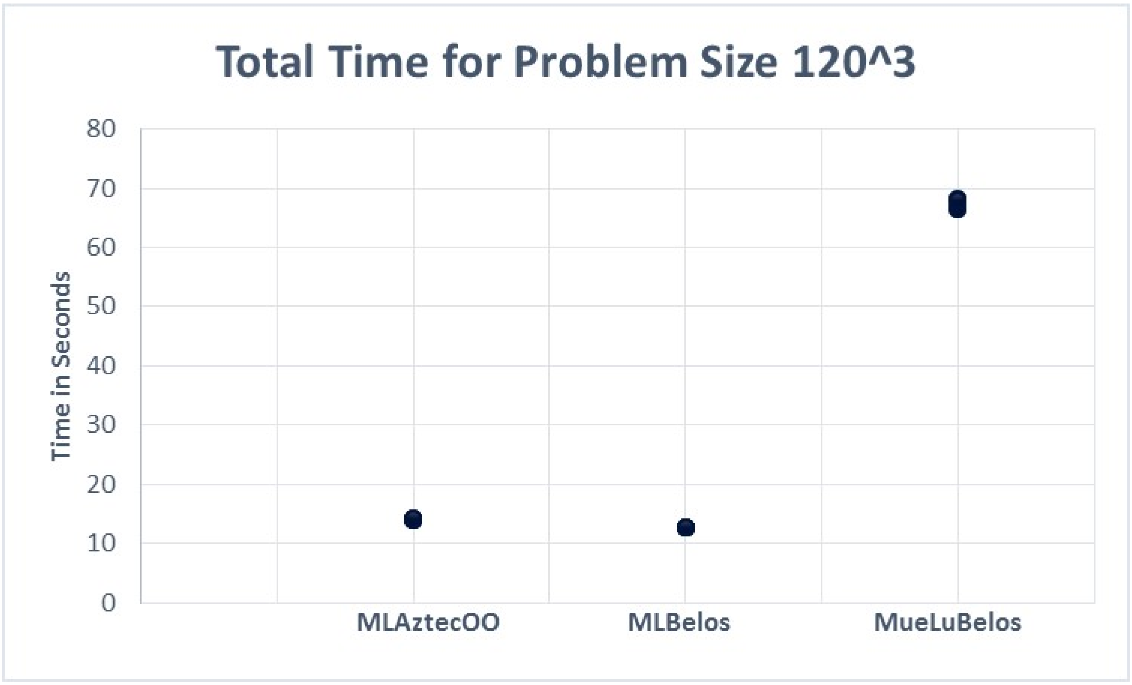

Figure 16: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 120^3. The measured time is total time to run including problem setup and solving. This plot shows the range in time performance for each file and the comparative time to complete between each file.

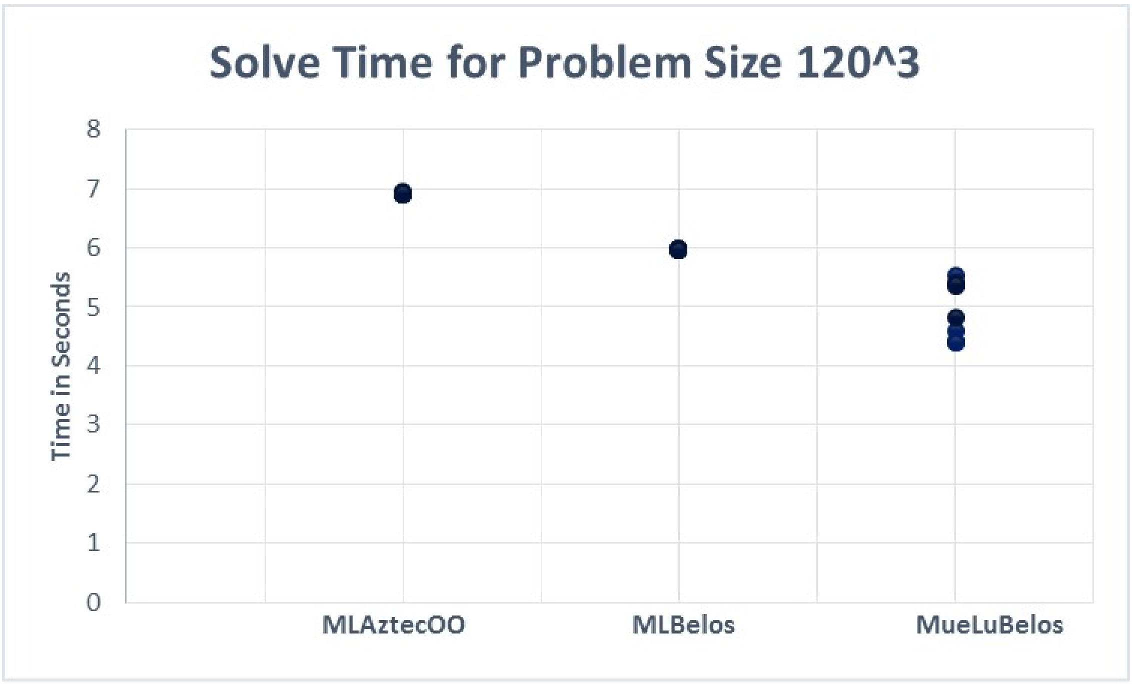

Figure 17: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 120^3. The measured time is the amount of time taken to solve the problem. Time to setup is not included. This plot shows the range in solving time performance for each file and the comparative time to solve between each file.

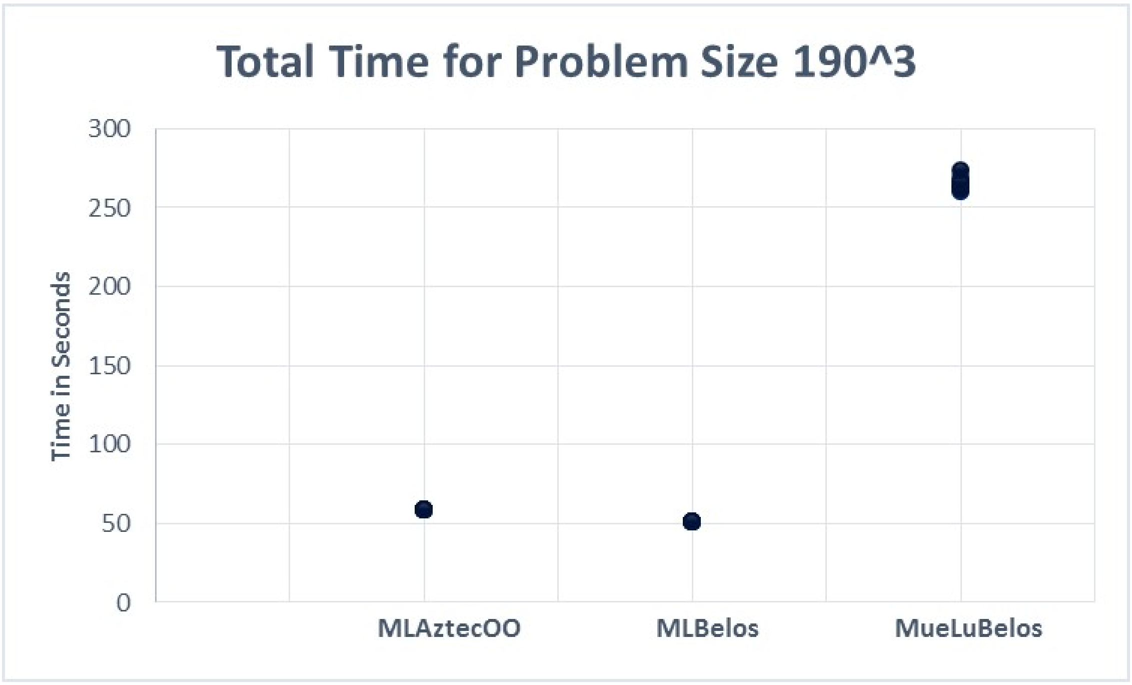

Figure 18: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 190^3. The measured time is total time to run including problem setup and solving. This plot shows the range in time performance for each file and the comparative time to complete between each file.

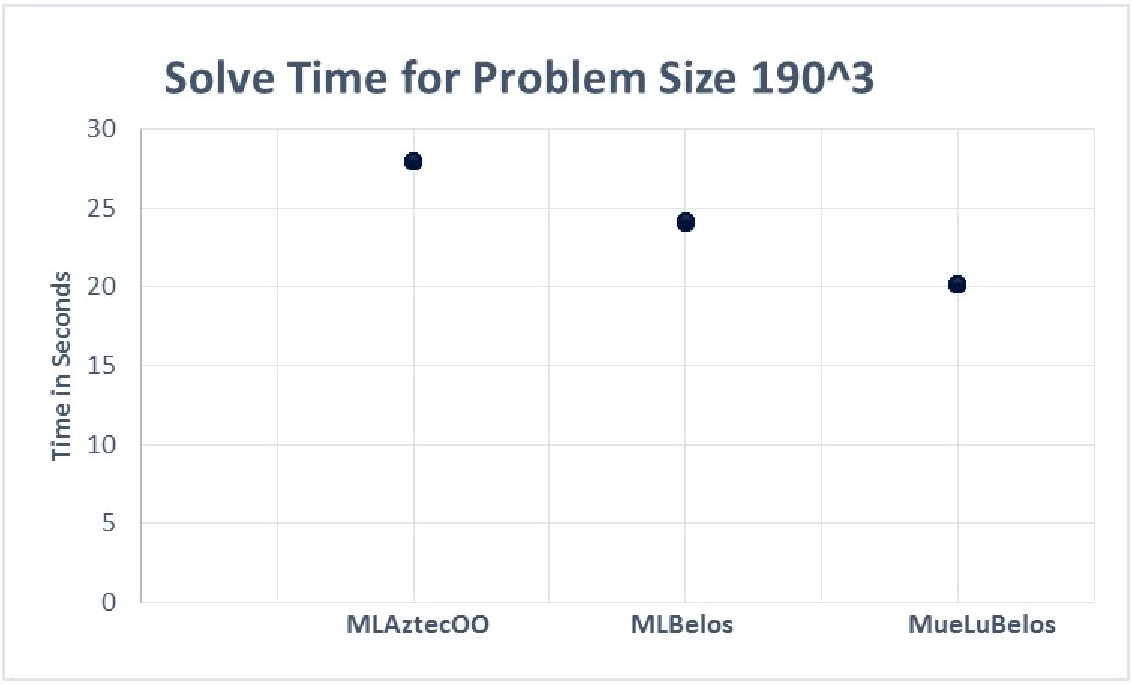

Figure 19: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 190^3. The measured time is the amount of time taken to solve the problem. Time to setup is not included. This plot shows the range in solving time performance for each file and the comparative time to solve between each file.

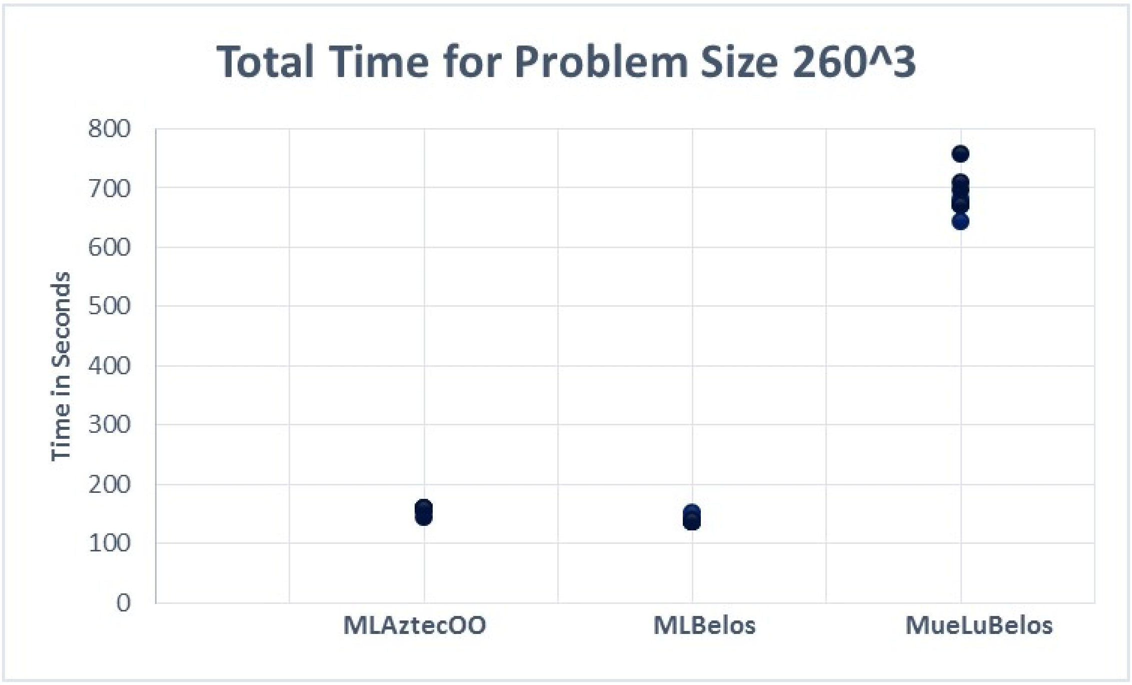

Figure 20: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 260^3. The measured time is total time to run including problem setup and solving. This plot shows the range in time performance for each file and the comparative time to complete between each file.

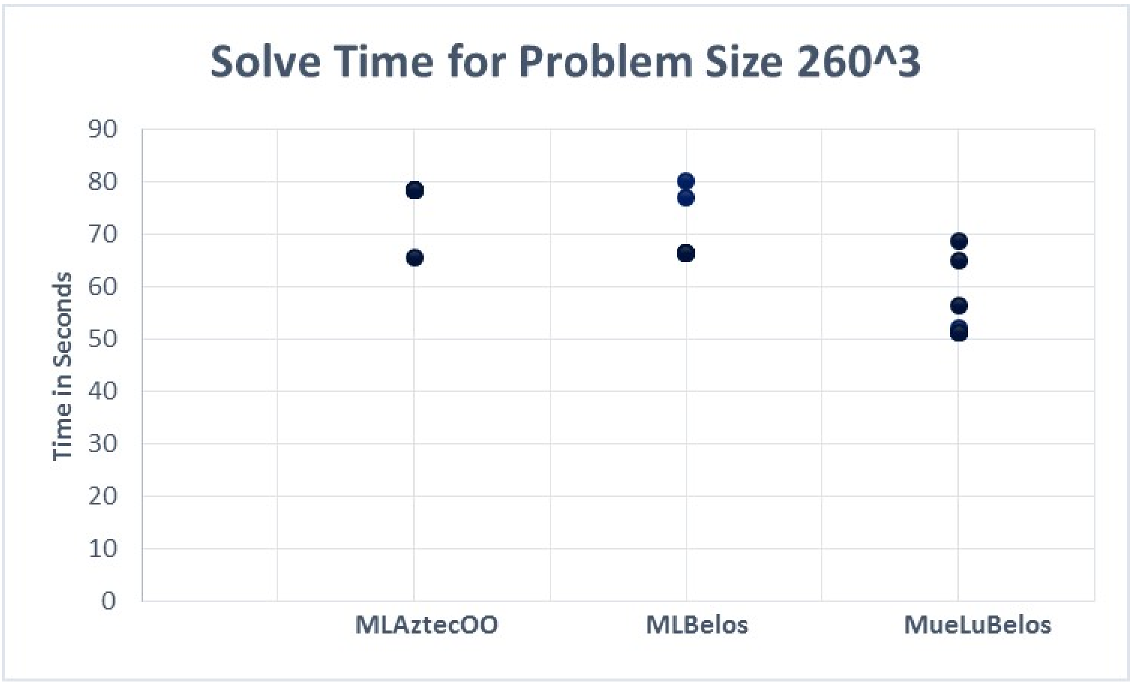

Figure 21: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 260^3. The measured time is the amount of time taken to solve the problem. Time to setup is not included. This plot shows the range in solving time performance for each file and the comparative time to solve between each file.

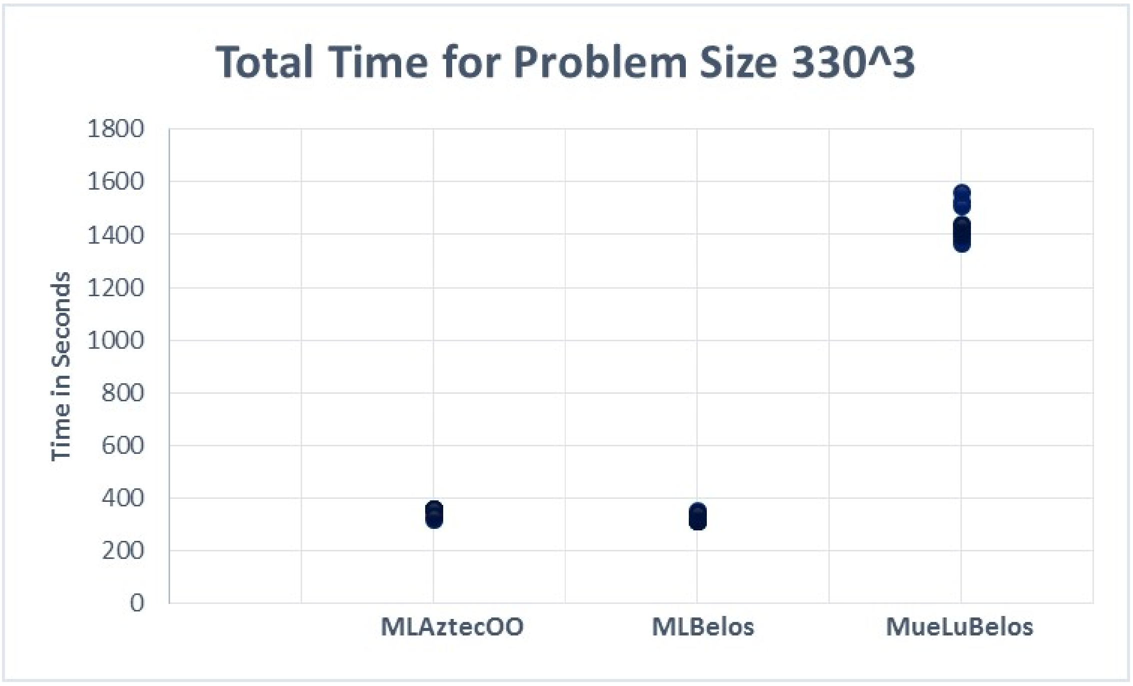

Figure 22: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 330^3. The measured time is total time to run including problem setup and solving. This plot shows the range in time performance for each file and the comparative time to complete between each file.

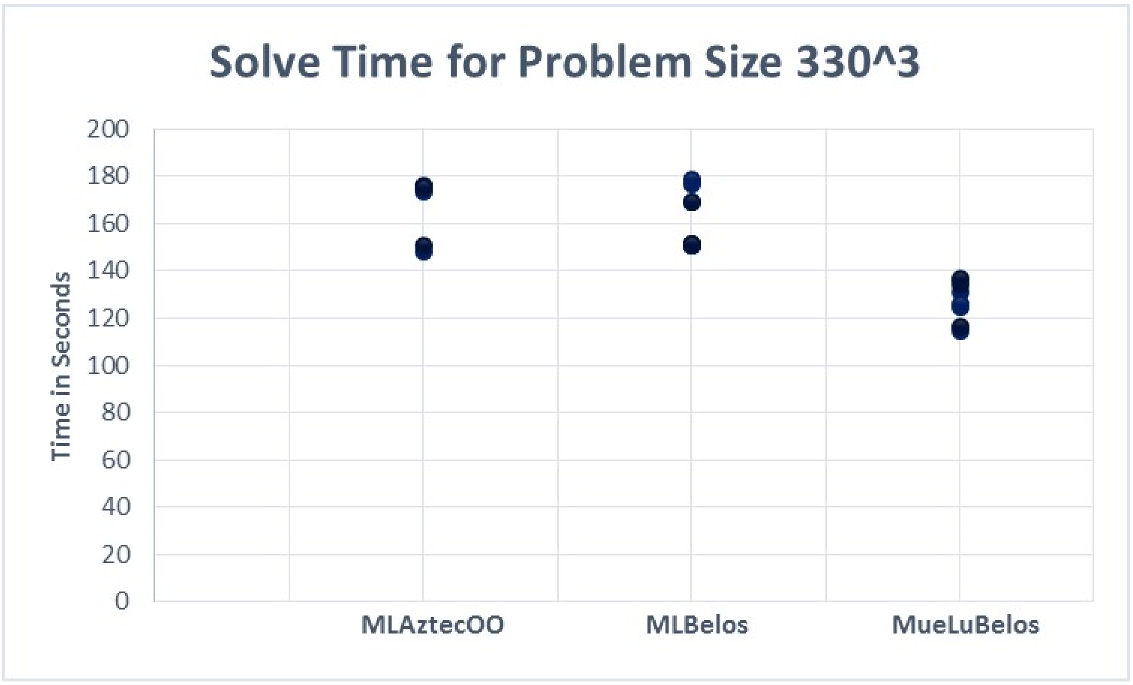

Figure 23: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 330^3. The measured time is the amount of time taken to solve the problem. Time to setup is not included. This plot shows the range in solving time performance for each file and the comparative time to solve between each file.

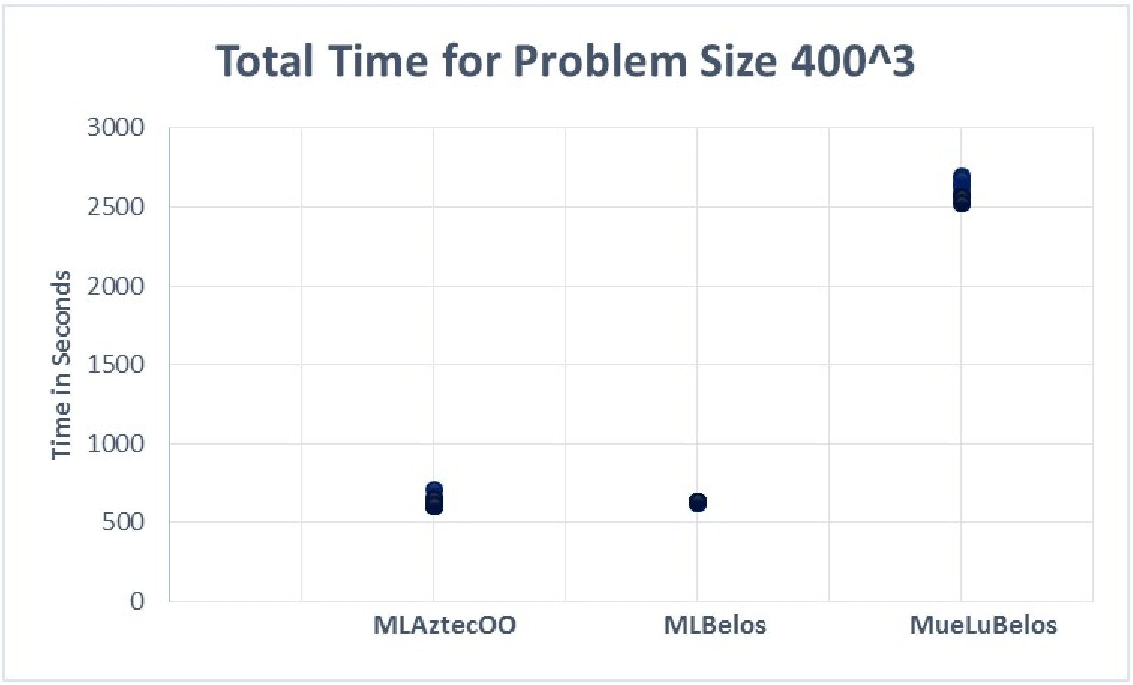

Figure 24: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 400^3. The measured time is total time to run including problem setup and solving. This plot shows the range in time performance for each file and the comparative time to complete between each file.

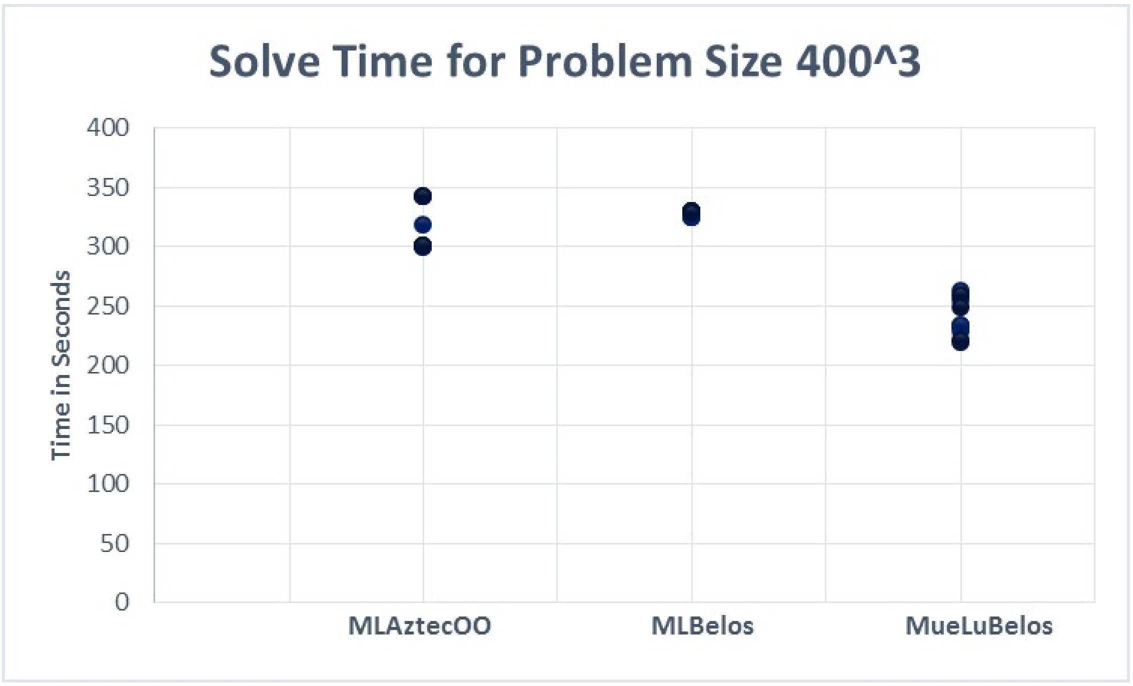

Figure 25: A scatterplot of 10 time trials for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp with a problem size of 400^3. The measured time is the amount of time taken to solve the problem. Time to setup is not included. This plot shows the range in solving time performance for each file and the comparative time to solve between each file.

Next, we analyze the results of the proposed method of updating the deal.II software. Overall, there is a trend of Belos performing slightly better as a solver than AztecOO. Further, MueLu paired with Belos on every problem size performed better than the other two implementations in terms of solving time. This trend can especially be seen in Figures 15, 17, 19, and 25. ML with AztecOO and ML with Belos performed very similarly concerning both solve time and total time. In general there was a trend of ML with Belos performing slightly better than ML with AztecOO in terms of both solving and total time. However, Figures 21, 23 ,24, and 25 show some instances in which ML with AztecOO performed better than ML with Belos on certain runs. These results are in line with what was expected. However, concerning total time, MueLu paired with Belos took almost five times as long to complete as either of the other two implementations. When looking at the source code it became clear what was causing the significant increase in time. The translation of the Epetra matrix to an Xpetra object (lines 104-110 of MueLuBelos.cpp in Appendix) is being executed one row at a time in a for loop. As the problem sizes increase, this very quickly becomes an issue. Some form of parallelism or a better method needs to be implemented to resolve this issue before deal.II can realistically implement the proposed method of integrating MueLu into its software.

7 Conclusion

1. **Performance of Finite Methods**

   This work confirmed that finite numerical methods find adequate approximations to traditional analytical solutions to differential equations. Further, it can be noted that the finite difference, forward in time scheme is not the most efficient method as the number of time steps required rapidly increases as the discretization of the problem space becomes more refined. This results in The finite element method using the Crank-Nicolson System converged to the analytical solution much more quickly than the finite difference method and was able to give a better approximation with a coarser discretization of the problem space than was achieved using the finite difference method.

2. **Proposed Method of Updating the deal.II Software**

   While, the small scale examples showed decrease in performance for total time when utilizing MueLu over ML, I predict that with added parallelism in the set up phase and with larger, more complex examples, time performance will increase. There is currently too much overhead created by translating the objects into Xpetra objects. Specifically, I believe that the current method of transferring the values of the matrix one row at a time is contributing the most to the total run time. Looking at just the time used for solving the problem shows that MueLu combined with Belos does perform better than ML and AztecOO together or ML and Belos together.

This thesis explored methods of solving partial differential equations that can be implemented with a computer. As computers cannot handle infinite amounts of points on a solution, it is necessary to discretize the solution space and solve using a finite numerical method. This work was able to show the accuracy of several such methods on the one-dimensional heat equation with Dirichlet boundary conditions. Further, a method of updating the deal.II software was proposed and its potential benefits were explored by analyzing time performance on small scale examples.

7.1 Future Work

In the future, a method should be developed for decreasing the overhead time created from translating the Epetra matrix to an Xpetra matrix. As the computation inside the current for loop accomplishing this transition is independent, I believe that this could easily be parallelized in order to increase performance. Further, if a parallel or distributed method were implemented in the Xpetra class to translate Epetra or Tpetra matrices, I believe that even better performance could be seen.

The next step along this course of work would be to implement the suggested changes in the deal.II software. As shown in the small scale examples, the code needs to be written so as to encapsulate or wrap the existing objects as Xpetra objects so that they are compatible with MueLu preconditioners. As hardware improves in accordance with Moore's Law, it is important to focus on maintaining and improving software as new technologies emerge and hardware improves.

References

[1] The trilinos project. trilinos.org. both documentation of and source code for the Trilinos Project.

[2] W. Bangerth, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and T. D. Young. The deal.ii library, version 8.1. *arXiv preprint, http://arxiv.org/abs/1312.2266v4*, 2013.

[3] John R. Cannon. *The One-Dimensional Heat Equation.* Addison-Wesley Publishing Company, Inc., 1984.

[4] D. DeTurck. Math 241: Solving the heat equation. 2012. Course notes for Math 241 at University of Pennsylvania.

[5] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* John Hopkins University Press, 2012.

[6] G. Recktenwald. Heatcn.m. Matlab code implementing the Crank-Nicolson scheme for solving the 1D heat equation.

[7] G. Recktenwald. Finite-difference approximations to the heat equation. 2011.

[8] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, 2003.

[9] P. Seshaiyar. Finite-difference method for the 1d heat equation. 2012. Course notes for Math679 at George Mason University.

[10] Griffiths D.V. Margetts L. Smith, I.M. *Programming the Finite Element Method.* John Wiley & Sons, 2013.

[11] R. Vichnevetsky. *Computer Methods for Partial Differential Equations, Volume 1.* Prentice-Hall, Inc., 1981.

[12] Taylor R. L. Zhu J. Z. Zienkiewicz, O. C. *Finite Element Method : Its Basis and Fundamentals.* Butterworth-Heinemann, 2005.

8 Appendix

8.1 Computing Architecture

Each node of Melchior consists of 12 cores each with 2 threads. Each thread state can be defined as follows:

```
Mel Nodes 0,1,2,4,5,6,7

processor        : 23
vendor_id        : GenuineIntel
cpu family       : 6
model            : 45
model name       : Intel(R) Xeon(R) CPU E5-2420 0 @ 1.90GHz
stepping         : 7
microcode        : 1808
cpu MHz          : 1200.000
cache size       : 15360 KB
physical id      : 1
siblings         : 12
core id          : 5
cpu cores        : 6
apicid           : 43
initial apicid   : 43
fpu              : yes
fpu_exception    : yes
cpuid level      : 13
wp               : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
     pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
    rdtscp lm constant_tsc arch_perfmon pebs bts rep_good xtopology nonstop_tsc
    aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr
    pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx
    lahf_lm ida arat epb xsaveopt pln pts dts tpr_shadow vnmi flexpriority ept vpid
bogomips         : 3793.04
clflush size     : 64
cache_alignment  : 64
address sizes    : 46 bits physical, 48 bits virtual
power management:
```

8.2 MLAztecOO.cpp Source Code

```
/*
* MLAztecOO.cpp
* Taken from Trilinos tutorial at https://code.google.com/p/trilinos/wiki/MLAztecOO
*
*/

//
// Use ML to build a smoothed aggregation multigrid operator.
// Use the operator as a black-box preconditioner in AztecOO's CG.
//
#include "Epetra_ConfigDefs.h"
#ifdef HAVE_MPI
#  include "mpi.h"
#  include "Epetra_MpiComm.h"
#else
#  include "Epetra_SerialComm.h"
#endif
#include "Epetra_Map.h"
#include "Epetra_Vector.h"
#include "Epetra_RowMatrix.h"
#include "Epetra_CrsMatrix.h"
#include "Epetra_LinearProblem.h"
#include "Epetra_Time.h"
#include "AztecOO.h"

// The ML include file required when working with Epetra objects.
#include "ml_epetra_preconditioner.h"

#include "Trilinos_Util_CrsMatrixGallery.h"

 using namespace Teuchos;
 using namespace Trilinos_Util;
 using namespace std;

#include <iostream>
#include <sys/time.h>
#include <time.h>

 int
 main (int argc, char *argv[])
 {

  struct timeval tim;
  gettimeofday(&tim, NULL);
  double t1=tim.tv_sec+(tim.tv_usec/1000000.0);


#ifdef EPETRA_MPI
  MPI_Init (&argc,&argv);
  Epetra_MpiComm Comm (MPI_COMM_WORLD);
#else
  Epetra_SerialComm Comm;
#endif

  Epetra_Time Time(Comm);

  // Initialize a Gallery object, for generating a 3-D Laplacian
  // matrix distributed over the given communicator Comm.
  CrsMatrixGallery Gallery("laplace_3d", Comm);

  Gallery.Set("problem_size", 1728000);

  // Get pointers to the generated matrix and a test linear problem.
  Epetra_RowMatrix* A = Gallery.GetMatrix();

  Epetra_LinearProblem* Problem = Gallery.GetLinearProblem();

  // Construct an AztecOO solver object for this problem.
  AztecOO solver (*Problem);

  // Create the preconditioner object and compute the multilevel hierarchy.
  ML_Epetra::MultiLevelPreconditioner * MLPrec =
    new ML_Epetra::MultiLevelPreconditioner(*A, true);

  // Tell AztecOO to use this preconditioner.
  solver.SetPrecOperator(MLPrec);

  // Tell AztecOO to use CG to solve the problem.
  solver.SetAztecOption(AZ_solver, AZ_cg);

  // Tell AztecOO to output status information every iteration
  // (hence the 1, which is the output frequency in terms of
  // number of iterations).
  solver.SetAztecOption(AZ_output, 1);
```

```

// Maximum number of iterations to try.
int Niters = 150;
// Convergence tolerance.
double tol = 1e-10;

//getting time after setting up problem
gettimeofday(&tim, NULL);
double t2=tim.tv_sec+(tim.tv_usec/1000000.0);

// Solve the linear problem.
solver.Iterate (Niters, tol);

//getting time after solver completes
gettimeofday(&tim, NULL);
double t3=tim.tv_sec+(tim.tv_usec/1000000.0);

// Print out some information about the preconditioner
if (Comm.MyPID() == 0)
  cout << MLPrec->GetOutputList();

// We're done with the preconditioner now, so we can deallocate it.
delete MLPrec;

// Verify the solution by computing the residual explicitly.
double residual = 0.0;
double diff = 0.0;
Gallery.ComputeResidual (&residual);
Gallery.ComputeDiffBetweenStartingAndExactSolutions (&diff);

// The Epetra_Time object has been keeping track of elapsed time
// locally (on this MPI process).  Take the min and max globally
// to find the min and max elapsed time over all MPI processes.
double myElapsedTime = Time.ElapsedTime ();
double minElapsedTime = 0.0;
double maxElapsedTime = 0.0;
(void) Comm.MinAll (&myElapsedTime, &minElapsedTime, 1);
(void) Comm.MaxAll (&myElapsedTime, &maxElapsedTime, 1);

if (Comm.MyPID()==0) {
  const int numProcs = Comm.NumProc ();
  cout << "||b-Ax||_2 = " << residual << endl
```

```
          << "||x_exact - x||_2 = " << diff << endl
          << "Min total time (s) over " << numProcs << " processes: "
          << minElapsedTime << endl
          << "Max total time (s) over " << numProcs << " processes: "
          << maxElapsedTime << endl;
  }

  //printing out time data
  cout <<  t3-t1 << " seconds elapsed" << endl;
  cout <<  t3-t2 << " seconds elapsed solving" << endl;

#ifdef EPETRA_MPI
  MPI_Finalize() ;
#endif
  return(EXIT_SUCCESS);
}
```

8.3 MLBelos.cpp Source Code

```
/*
* MLBelos.cpp
* by Emily Furst
* Modified from original Trilinos tutorial at https://code.google.com/p/trilinos/
    wiki/MLAztecOO
*
*/

//
// Use ML to build a smoothed aggregation multigrid operator.
// Use Belos as a solver
//
#include "Epetra_ConfigDefs.h"
#ifdef HAVE_MPI
#  include "mpi.h"
#  include "Epetra_MpiComm.h"
#else
#  include "Epetra_SerialComm.h"
#endif
#include "Epetra_Map.h"
#include "Epetra_Vector.h"
#include "Epetra_RowMatrix.h"
#include "Epetra_CrsMatrix.h"
```

```
#include "Epetra_LinearProblem.h"
#include "Epetra_Time.h"
#include "AztecOO.h"
#include "Teuchos_ParameterList.hpp"
#include "Teuchos_RCP.hpp"
#include "BelosLinearProblem.hpp"
#include "BelosBlockCGSolMgr.hpp"
#include "BelosEpetraAdapter.hpp"
#include "MueLu.hpp"


// The ML include file required when working with Epetra objects.
#include "ml_epetra_preconditioner.h"

#include "Trilinos_Util_CrsMatrixGallery.h"

using namespace Teuchos;
using namespace Trilinos_Util;
using namespace std;

#include <iostream>
#include <sys/time.h>
#include <time.h>

int
main (int argc, char *argv[])
{

  struct timeval tim;
  gettimeofday(&tim, NULL);
  double t1=tim.tv_sec+(tim.tv_usec/1000000.0);


#ifdef EPETRA_MPI
  MPI_Init (&argc,&argv);
  Epetra_MpiComm Comm (MPI_COMM_WORLD);
#else
  Epetra_SerialComm Comm;
#endif

  Epetra_Time Time(Comm);

```

```
// Initialize a Gallery object, for generating a 3-D Laplacian
// matrix distributed over the given communicator Comm.
CrsMatrixGallery Gallery("laplace_3d", Comm);

//problem size must be a perfect cube
Gallery.Set("problem_size", 1728000);

// Get pointers to the generated matrix and a test linear problem.

RCP<Epetra_RowMatrix> A = rcp (Gallery.GetMatrix(), false);

RCP<Epetra_MultiVector> LHS = rcp (Gallery.GetLinearProblem()->GetLHS(), false);
RCP<Epetra_MultiVector> RHS = rcp (Gallery.GetLinearProblem()->GetRHS(), false);




typedef Epetra_MultiVector              MV;
typedef Epetra_Operator                 OP;

RCP<Belos::LinearProblem<double,MV,OP> > problem = rcp (new Belos::LinearProblem<
    double,MV,OP>(A, LHS, RHS));

bool set = problem->setProblem();
TEUCHOS_TEST_FOR_EXCEPTION( ! set,
                      std::runtime_error,
                      "*** Belos::LinearProblem failed to set up correctly! ***");


// Create the preconditioner object and compute the multilevel hierarchy.
ML_Epetra::MultiLevelPreconditioner * MLPrec =
  new ML_Epetra::MultiLevelPreconditioner(*A, true);

//Begin to set up Belos solver

RCP<Belos::EpetraPrecOp> prec = rcp(new Belos::EpetraPrecOp(rcp(MLPrec, false)));
problem->setRightPrec (prec);


RCP<ParameterList> belosList = rcp (new ParameterList ());
belosList->set ("Block Size", 1);               // Blocksize to be used by
    iterative solver
```

```
belosList->set ("Maximum_Iterations", 150);    // Maximum number of iterations
    allowed
belosList->set ("Convergence_Tolerance", 1e-10);// Relative convergence tolerance
     requested
belosList->set ("Verbosity", Belos::Errors+Belos::Warnings+Belos::TimingDetails+
    Belos::FinalSummary+Belos::StatusTestDetails+Belos::IterationDetails );



Belos::BlockCGSolMgr<double,MV,OP> belosSolver (problem, belosList);


//retrieve time before solver computes and after setup
gettimeofday(&tim, NULL);
double t2=tim.tv_sec+(tim.tv_usec/1000000.0);


//Belos solver solves problem
Belos::ReturnType ret = belosSolver.solve();


//retrieve time after solving completed
gettimeofday(&tim, NULL);
double t3=tim.tv_sec+(tim.tv_usec/1000000.0);


// We're done with the preconditioner now, so we can deallocate it.
delete MLPrec;

// Verify the solution by computing the residual explicitly.
double residual = 0.0;
double diff = 0.0;
Gallery.ComputeResidual (&residual);
Gallery.ComputeDiffBetweenStartingAndExactSolutions (&diff);

// The Epetra_Time object has been keeping track of elapsed time
// locally (on this MPI process).  Take the min and max globally
// to find the min and max elapsed time over all MPI processes.
double myElapsedTime = Time.ElapsedTime ();
double minElapsedTime = 0.0;
double maxElapsedTime = 0.0;
(void) Comm.MinAll (&myElapsedTime, &minElapsedTime, 1);
```

```
  (void) Comm.MaxAll (&myElapsedTime, &maxElapsedTime, 1);

  cout<<endl;

  cout << "Parameter List: " << *belosList <<endl;

  if (Comm.MyPID()==0) {
    const int numProcs = Comm.NumProc ();
    cout << "||b-Ax||_2 = " << residual << endl
         << "||x_exact - x||_2 = " << diff << endl
         << "Min total time (s) over " << numProcs << " processes: "
         << minElapsedTime << endl
         << "Max total time (s) over " << numProcs << " processes: "
         << maxElapsedTime << endl;
    if (ret == Belos::Converged) {
      std::cout << "Belos converged." << std::endl;
    } else {
      std::cout << "Belos did not converge." << std::endl;
    }
  }


  cout <<  t3-t1 << " seconds elapsed" << endl;
  cout <<  t3-t2 << " seconds elapsed solving" << endl;

#ifdef EPETRA_MPI

  MPI_Finalize() ;
#endif

  return(EXIT_SUCCESS);
}
```

8.4 MueLuBelos.cpp Source Code

```
/*
* MueLuBelos.cpp
* by Emily Furst
* Modified from MLBelos.cpp and original Trilinos tutorial at https://code.google.
   com/p/trilinos/wiki/MLAztecOO
*
*/
```

```

//
//Use MueLu preconditioner with Belos solver to solve problem set up
//
#include "Epetra_ConfigDefs.h"
#ifdef HAVE_MPI
#  include "mpi.h"
#  include "Epetra_MpiComm.h"
#else
#  include "Epetra_SerialComm.h"
#endif
#include "Epetra_Map.h"
#include "Epetra_Vector.h"
#include "Epetra_RowMatrix.h"
#include "Epetra_CrsMatrix.h"
#include "Epetra_LinearProblem.h"
#include "Epetra_Time.h"
#include "Teuchos_ParameterList.hpp"
#include "Teuchos_RCP.hpp"
#include "BelosLinearProblem.hpp"
#include "BelosBlockCGSolMgr.hpp"
#include "BelosEpetraAdapter.hpp"
#include "MueLu.hpp"
#include "Xpetra_Vector.hpp"
#include "Xpetra_RowMatrix.hpp"
#include "Xpetra_CrsMatrix.hpp"
#include "Xpetra_EpetraCrsMatrix.hpp"

#include <iostream>

#include <Xpetra_MultiVectorFactory.hpp>


#include <MueLu_TrilinosSmoother.hpp> //TODO: remove

// Header files defining default types for template parameters.
// These headers must be included after other MueLu/Xpetra headers.
#include <MueLu_UseDefaultTypes.hpp>  // => Scalar=double, LocalOrdinal=int,
    GlobalOrdinal=int

#include <BelosConfigDefs.hpp>
#include <BelosLinearProblem.hpp>
```

```
#include <BelosBlockCGSolMgr.hpp>
#include <BelosXpetraAdapter.hpp>     // => This header defines Belos::XpetraOp
#include <BelosMueLuAdapter.hpp>      // => This header defines Belos::MueLuOp


// The ML include file required when working with Epetra objects.
#include "ml_epetra_preconditioner.h"

#include "Trilinos_Util_CrsMatrixGallery.h"

using namespace Teuchos;
using namespace Trilinos_Util;
using namespace std;
using namespace Xpetra;
using namespace MueLu;

#include <iostream>
#include <sys/time.h>
#include <time.h>

int
main (int argc, char *argv[])
{

  struct timeval tim;
  gettimeofday(&tim, NULL);
  double t1=tim.tv_sec+(tim.tv_usec/1000000.0);

#include <MueLu_UseShortNames.hpp>

#ifdef EPETRA_MPI
  MPI_Init (&argc,&argv);
  Epetra_MpiComm Comm (MPI_COMM_WORLD);
#else
  Epetra_SerialComm Comm;
#endif

  Epetra_Time Time(Comm);

  // Initialize a Gallery object, for generating a 3-D Laplacian
  // matrix distributed over the given communicator Comm.
  CrsMatrixGallery Gallery("laplace_3d", Comm);

  //problem size must be a perfect cube as working with 3-D Laplacian
  Gallery.Set("problem_size", 125000);

  // Get pointers to the generated matrix and a test linear problem.
  Epetra_CrsMatrix A = *(Gallery.GetMatrix());
  Epetra_CrsGraph graphA = A.Graph();

  //convert Epetra Matrix to Xpetra Matrix object

  RCP<const CrsGraph > rgraphA = toXpetra<int >(graphA);
  RCP<Matrix > rA = rcp(new CrsMatrixWrap(rgraphA));
  rA->setAllToScalar(0);
  rA->resumeFill();
  for(int i = 0; i < A.NumGlobalRows(); i++){
    int num = 0, * indices = new int[A.NumGlobalEntries(i)];
    double * values = new double[A.NumGlobalEntries(i)];
    A.ExtractGlobalRowCopy(i, A.NumGlobalCols(),num, values, indices);

    rA->replaceGlobalValues(i, arrayView<const int >(indices, num), arrayView<const
        double>(values, num));
  }
  rA->fillComplete();

  //retrieve LHS and RHS vectors

  Epetra_LinearProblem* Problem = Gallery.GetLinearProblem();
  RCP<Epetra_MultiVector> LHS = rcp(Problem->GetLHS(), false);
  RCP<Epetra_MultiVector> RHS = rcp(Problem->GetRHS(), false);

  //convert LHS and RHS Epetra vectors to Xpetra objects

  RCP<MultiVector > rLHS = toXpetra<int >(LHS);
  RCP<MultiVector > rRHS = toXpetra<int >(RHS);


  //setting up MueLu preconditioner and Belos solvers
  FactoryManager M;

  RCP<Hierarchy> H = rcp(new Hierarchy(rA));

  H->setVerbLevel(Teuchos::VERB_HIGH);

  RCP<Factory> AcFact = rcp(new RAPFactory());
  M.SetFactory("A", AcFact);



  H->Setup(M);

  typedef MultiVector MV;
  typedef Belos::OperatorT<MV> OP;

  RCP<OP> belosOp = rcp(new Belos::XpetraOp<SC, LO, GO, NO>(rA));
  RCP<OP> belosPrec = rcp(new Belos::MueLuOp<SC, LO, GO, NO>(H));

  RCP< Belos::LinearProblem<SC, MV, OP> > belosProblem = rcp(new Belos::
      LinearProblem<SC, MV, OP>(belosOp, rLHS, rRHS));
  belosProblem->setLeftPrec(belosPrec);

  bool set = belosProblem->setProblem();
  TEUCHOS_TEST_FOR_EXCEPTION( ! set,
                          std::runtime_error,
                          "*** Belos::LinearProblem failed to set up correctly! ***");
  int maxIts = 150;
  double tol = 1e-10;
  Teuchos::ParameterList belosList;
  belosList.set("Maximum Iterations", maxIts); // Maximum number of iterations
      allowed
  belosList.set("Convergence Tolerance", tol); // Relative convergence tolerance
      requested
  belosList.set("Verbosity", Belos::Errors + Belos::Warnings + Belos::TimingDetails
       + Belos::StatusTestDetails);

  // Create an iterative solver manager
  RCP< Belos::SolverManager<SC, MV, OP> > solver = rcp(new Belos::BlockCGSolMgr<SC,
       MV, OP>(belosProblem, rcp(&belosList, false)));


  //retrieve time after setup and before solve
  gettimeofday(&tim, NULL);
  double t2=tim.tv_sec+(tim.tv_usec/1000000.0);

  // Perform solve

```

```cpp
Belos::ReturnType ret = solver->solve();

//retrieve time after solve
gettimeofday(&tim, NULL);
double t3=tim.tv_sec+(tim.tv_usec/1000000.0);

// Get the number of iterations for this solve.
std::cout << "Number of iterations performed for this solve: " << solver->
    getNumIters() << std::endl;



// Verify the solution by computing the residual explicitly.
double residual = 0.0;
double diff = 0.0;
Gallery.ComputeResidual (&residual);
Gallery.ComputeDiffBetweenStartingAndExactSolutions (&diff);

// The Epetra_Time object has been keeping track of elapsed time
// locally (on this MPI process).  Take the min and max globally
// to find the min and max elapsed time over all MPI processes.
double myElapsedTime = Time.ElapsedTime ();
double minElapsedTime = 0.0;
double maxElapsedTime = 0.0;
(void) Comm.MinAll (&myElapsedTime, &minElapsedTime, 1);
(void) Comm.MaxAll (&myElapsedTime, &maxElapsedTime, 1);

cout<<endl;


if (Comm.MyPID()==0) {
  const int numProcs = Comm.NumProc ();
  cout << "||b-Ax||_2 = " << residual << endl
       << "||x_exact - x||_2 = " << diff << endl
       << "Min total time (s) over " << numProcs << " processes: "
       << minElapsedTime << endl
       << "Max total time (s) over " << numProcs << " processes: "
       << maxElapsedTime << endl;
  if (ret == Belos::Converged) {
    std::cout << "Belos converged." << std::endl;
  } else {
    std::cout << "Belos did not converge." << std::endl;
    }
  }

  //print out time results
  cout <<  t3-t1 << "␣seconds␣elapsed" << endl;
  cout <<  t3-t2 << "␣seconds␣elapsed␣solving" << endl;

#ifdef EPETRA_MPI

  MPI_Finalize() ;
#endif
  return(EXIT_SUCCESS);
}
```

8.5 Makefile for MLAztecOO.cpp

Note Makefiles for MLAztecOO.cpp, MLBelos.cpp, and MueLuBelos.cpp are essentially the same with only the filename and executable name changed wherever they appear in the Makefile (lines 55,56,59,60).

```
 1 # CMAKE File for "MyApp" application building against an installed Trilinos
 2
 3 #This file is an adaptation of the CMakeLists.txt file that was converted from
 4 #the buildAgainstTrilinos example. This Makefile was designed to be used in a
 5 #flat directory structure. If you would like to run this example you will need
 6 #put this file and src_file.cpp, src_file.hpp, main_file.cpp from
 7 #buildAgainstTrilinos into a new directory. You will then need to set the
 8 #environment variable MYAPP_TRILINOS_DIR to point to your base installation of
 9 #Trilinos. Note that this example assumes that the installation of Trilinos that
10 #you point to has Epetra enabled.
11
12 # Get Trilinos as one entity
13  include $(MYAPP_TRILINOS_DIR)include/Makefile.export.Trilinos
14
15 # Make sure to use same compilers and flags as Trilinos
16 CXX=$(Trilinos_CXX_COMPILER)
17 CC=$(Trilinos_C_COMPILER)
18 FORT=$(Trilinos_Fortran_COMPILER)
19
20 CXX_FLAGS=$(Trilinos_CXX_COMPILER_FLAGS) $(USER_CXX_FLAGS)
21 C_FLAGS=$(Trilinos_C_COMPILER_FLAGS) $(USERC_FLAGS)
22 FORT_FLAGS=$(Trilinos_Fortran_COMPILER_FLAGS) $(USER_FORT_FLAGS)
```

```

INCLUDE_DIRS=$(Trilinos_INCLUDE_DIRS) $(Trilinos_TPL_INCLUDE_DIRS)
LIBRARY_DIRS=$(Trilinos_LIBRARY_DIRS) $(Trilinos_TPL_LIBRARY_DIRS)
LIBRARIES=$(Trilinos_LIBRARIES) $(Trilinos_TPL_LIBRARIES)

LINK_FLAGS=$(Trilinos_EXTRA_LD_FLAGS)

#just assuming that epetra is turned on.
DEFINES=-DMYAPP_XPETRA
DEFINES=-DMYAPP_EPETRA
DEFINES=-DMYAPP_TPETRA
DEFINES=-DMYAPP_GALERI
DEFINES=-DMYAPP_STRATIMIKOS

default: print_info MLEx.exe

# Echo trilinos build info just for fun
print_info:
	@echo "\nFound Trilinos!  Here are the details: "
	@echo "   Trilinos_VERSION = $(Trilinos_VERSION)"
	@echo "   Trilinos_PACKAGE_LIST = $(Trilinos_PACKAGE_LIST)"
	@echo "   Trilinos_LIBRARIES = $(Trilinos_LIBRARIES)"
	@echo "   Trilinos_INCLUDE_DIRS = $(Trilinos_INCLUDE_DIRS)"
	@echo "   Trilinos_LIBRARY_DIRS = $(Trilinos_LIBRARY_DIRS)"
	@echo "   Trilinos_TPL_LIST = $(Trilinos_TPL_LIST)"
	@echo "   Trilinos_TPL_INCLUDE_DIRS = $(Trilinos_TPL_INCLUDE_DIRS)"
	@echo "   Trilinos_TPL_LIBRARIES = $(Trilinos_TPL_LIBRARIES)"
	@echo "   Trilinos_TPL_LIBRARY_DIRS = $(Trilinos_TPL_LIBRARY_DIRS)"
	@echo "   Trilinos_BUILD_SHARED_LIBS = $(Trilinos_BUILD_SHARED_LIBS)"
	@echo "End of Trilinos details\n"

# run the given test
test: MLEx.exe input.xml
	./MLEx.exe

# build the
MLEx.exe: MLAztecOO.cpp
	$(CXX) $(CXX_FLAGS) MLAztecOO.cpp -o MLEx.exe $(LINK_FLAGS) $(INCLUDE_DIRS) $(DEFINES) $(LIBRARY_DIRS) $(LIBRARIES)

libmyappLib.a: src_file.o
	$(Trilinos_AR) cr libmyappLib.a src_file.o
```

```

src_file.o:
	$(CXX) -c $(CXX_FLAGS) $(INCLUDE_DIRS) $(DEFINES) MLAztecOO.cpp

.PHONY: clean
clean:
	rm -f *.o *.a *.exe
```

8.6 Trilinos cmake Script

The following is the cmake script used to install Trilinos based off of a version of the software downloaded from the public repository.

```
Trilinos Cmake Script

/opt/cmake-2.8.12.2/bin/cmake -D CMAKE_BUILD_TYPE:STRING=DEBUG -D
    Trilinos_ENABLE_TESTS:BOOL=OFF -D Trilinos_ASSERT_MISSING_PACKAGES=OFF -D
    Trilinos_ENABLE_ALL_PACKAGES=ON -D TPL_ENABLE_MPI:BOOL=ON -D
    Trilinos_ENABLE_MueLu=ON -D BUILD_SHARED_LIBS:BOOL=ON -D CMAKE_CXX_FLAGS="-g -
    O3" -D CMAKE_C_FLAGS="-g -O3" -D CMAKE_FORTRAN_FLAGS="-g -O5" -D
    Trilinos_EXTRA_LINK_FLAGS="-lgfortran" -D CMAKE_INSTALL_PREFIX:PATH=/usr/people
    /research/eafurst/trilinos-date /usr/people/research/eafurst/trilinos-11.8.1-
    Source/publicTrilinos
```

8.7 deal.II Install Notes

The following are the commands used to install a version of deal.II downloaded from the deal.II website.

```
mkdir build
cd build

cmake -DCMAKE_INSTALL_PREFIX=/path/to/install/dir ../deal.II -DDEAL_II_WITH_MPI=ON
    -DTRILINOS_DIR=/path/to/trilinos


make install
make test
```