2008-01-01

A Distributed Reconstruction of EKG Signals

Gabriel Cordova
University of Texas at El Paso, gcordova@miners.utep.edu

A DISTRIBUTED RECONSTRUCTION OF EKG SIGNALS

GABRIEL CORDOVA

Department of Electrical and Computer Engineering

APPROVED:

_____
David H. Williams, Ph.D., Chair

_____
Patricia A. Nava, Ph.D., Department Chair

_____
Rodrigo Romero, Ph.D.

_____
Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School

To Rachel and all of my family and friends for all of their support

A DISTRIBUTED RECONSTRUCTION OF EKG SIGNALS

BY

GABRIEL CORDOVA, BSEE

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

THE UNIVERSITY OF TEXAS AT EL PASO

December 2008

Acknowledgements

It has been a privilege to obtain an education from the University of Texas at El Paso. The experiences and lessons learned have shaped the person that I am today. I would like to thank God for imparting to me the knowledge to be successful and the motivation to move forward. This thesis would not have been possible without the love and support from my beautiful new wife, Rachel. I would also like to thank my mother, Gabriela Palma, for being the best role model and instilling in me the value of honesty and hard work.

I would like to thank Dr. David H. Williams for his encouragement to continue with my graduate studies. He provided me with guidance and advice not only in my academic career but also in life. Special thanks to Dr. Patricia Nava who was always willing to listen and offer a word of advice. For her interest and dedication to students, and for the generous support that she provided. I would also thank Dr. Rodrigo Romero for his willingness to be a part of my thesis committee. I would also like to express my appreciation to Eduardo Morales for his patience and willingness to help; for sharing his time and ideas.

Thanks to my fellow UNIX Admins. My appreciation goes out to Nito for help as it related to my thesis and general computing. Thanks to Damian and Jon for the endless supply of Reese's. Thank you to all of the faculty and staff of the Electrical and Computer Engineering Department at the University of Texas at El Paso.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0709438. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Network switches were provided by Cisco Systems, Inc.

Abstract

In this thesis the parallel computing methodology is applied to an algorithm used in the reconstruction of electrocardiographic (EKG) measurements. The reconstructions are being performed to obtain a better understanding of the source and behavior of the electrical activity that generates the EKG measurements. The contribution of this thesis is to identify and eliminate inefficiencies present in the current reconstruction algorithm. Additionally, this thesis reduces the computation times of the EKG reconstruction by applying distributed computing through the use of Remote Procedure Calls (RPC). Lastly, it provides an analysis of the speed-up and efficiency of the distribution implemented using parallel processes and threads.

The tests conducted show that addressing inefficiencies in the original algorithm provided a decrease in computation times by a factor of 40. Additionally, it showed that using multi-threading to distribute the tasks in the client side of the RPC distribution was more effective than using multiple processes. The results also show that implementing threads to parallelize the server side code of the RPC distribution was a hindrance rather than a benefit to the reconstruction problem as it yielded slower run times.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 History

From their inception, computers were designed to automate tasks in order to streamline routine processes. Whether they were large-scale scientific computers used for computations or commercial computers used for sorting and printing, early computers helped to reduce human error while being convenient in their ease of use [1]. However, despite the convenience that computers have provided, they have always been limited by the available technology of their time. These limitations included, but where not limited to, memory capacity, disk space, number of I/O and processing speeds [1]. Initially, small memory capacities limited the size of programs that could be written and as a result limited the extent of problems that such programs could address. Similarly, limitations in storage space and processing speeds have restricted the types of applications that could be developed. Fortunately, human ingenuity coupled with the promise from the success of these early machines has led to the development of work-around solutions. These solutions not only contributed to the development of technology, but allowed it time to develop. For example, to address the limited availability of memory the concepts of swapping and virtual memory were developed, which allowed more and larger programs to run within the limited memory [1]. Also, a variety of alternatives for file storage have evolved over the years, while processing speeds have grown at increasing rates. The leaps in computational power within the last decade have been phenomenal allowing computers to handle jobs never thought possible with previous technologies. For example, several years back a typical personal computer consisted of a couple of hundred megabytes of memory, a 10 gigabyte hard drive, and a single processor limited to top speeds in the megahertz. Nowadays, average computers are

equipped with a couple gigabytes of main memory, hard drives that exceed 100 gigabytes, and a multi-core processor running in excess of 2 gigahertz. These advances have allowed for the development of applications that extend much further than the original computational machines of the past to include complex simulations capable of diagnosing disease, advanced image and video processing and a variety of simulations that very closely approximate real behaviors. Despite advances, there continues to be applications that push even the most up to date machines to their limits. Often the limits of the more advanced systems remain the same as their older counterparts but to a larger scale. Therefore, old techniques continue to be adapted to provide superior computing power. However, the increasing complexity and size of the problems being addressed by current systems will often cause long delays before useful output is generated. A current example of such an application is that of reconstructing electrocardiogram measurements which is one the current undertakings of the Bio-potentials Group (BPG) at the University of Texas at El Paso (UTEP) spearheaded by Eduardo Morales [2]. Their attempts at reconstructing these measurements have been hindered due to the consumption of resources by their algorithms leading to long run times (in the vicinity of 5 minutes per patient reconstruction).

1.2 Electrocardiography

Measuring the electrical activity generated by the heart was first introduced in the late 1880's when the first human electrocardiogram (EKG) was recorded. Although the techniques have evolved over the years, electrocardiography has been used in diagnostics from the onset of the science when only three leads, or electrical connections used to measure potential difference, were employed [3]. Currently it is common place to use a 12-lead system when performing an EKG, but redundancy in the output of the different leads has allowed for measuring fewer leads while still outputting values for 12 leads [3].

Although different methods of studying the heart's activity have been developed, the EKG continues to play an important role as a tool for physicians for cardiovascular assessment [2, 3]. The main reasons for the continued use of EKG is that it is a fast, low-cost, and non-invasive way of obtaining information about the condition of the heart. It provides doctors with information on cardiac rhythm, presence or absence of conduction defects and the degree of myocardial damage following an infarction [3]. Through its use doctors can diagnose certain heart conditions and take preventative action to save the lives of their patients.

Because of the importance that EKG's play in the diagnosis of heart disease, there is a great interest in further understanding the origin of the electrical activity measured by an EKG. The UTEP BPG intends to increase their understanding of the electrical activity of the heart by developing a reconstruction of the EKG measurements by means of a mathematical model [2]. They are motivated because obtaining this understanding can result in improvements to current techniques for identification of cardiac abnormalities [2]. The reconstruction is based on a dipole model of the heart which is applied to a set of precordial leads. The dipole model consists of more than twenty-thousand sets of values. Each set consisting of 3 vector values for 6

different leads. Although, the dipole model of the heart contains values for all 12 leads, the BPG has chosen to only use six for their initial reconstruction purposes. However, their algorithm is designed in such a way that if necessary more leads can be added to their calculations.

The current algorithm used for the reconstruction uses a Matlab script which loops through all sets of 20,571 dipole model values. For each set of values, the dipole moment is calculated using QR factorization. After the dipole moment is obtained the reconstruction is attempted using the current set of dipole model values. The reconstructed result is then compared against the original measured EKG values using RMS error calculations, and stores the error for that particular set of dipole model values in an array. After all dipole model values have been iterated through, the smallest error is found and returned as the solution for the reconstruction with the smallest error.

Clearly, some of the computations being performed must occur in a serial manner. That is, there is no feasible way to reconstruct the EKG before the dipole moment is computed. Likewise, the error cannot be calculated before the signal is reconstructed. However, if these three steps are regarded as one then it can be said that the single action consisting of, calculating the dipole moment, reconstructing the signal and calculating the error occurs 20,571 times. Each occurrence does not depend on the other, and thus this application is a perfect candidate to benefit greatly from a parallel implementation.

1.3 Parallel Computing

Traditionally programs have been designed to run serially. In other words, instructions are processed by the CPU one at a time in the order that they are received, but parallel processing allows for independent sections of code to be run concurrently. This method takes full advantage of systems with multiple processors or processors with multiple cores.

It appears evident that programs can benefit if written with a mindset of parallelism, but the truth is that not all programs are created equal. Some problems require serial operation due mainly to data dependencies, and also as a result of other special conditions such as atomic sections of code created to protect critical regions. Therefore, before any program is converted into a parallel application or before a parallel application is coded, the tradeoffs need to be weighed. A coder must determine how much the given application will benefit from parallelism, and if those benefits outweigh the costs of developing and maintaining a parallel application. The reasons for applying parallelism when it is appropriate are evident. Computation times can be reduced drastically, and the resources of the computer can be used to utmost efficiency. Additional benefits from parallel computing include the ability to solve larger problems, as well as reducing operating costs.

As indicated before, the reconstruction algorithm being implemented by the UTEP BPG is an excellent candidate to be implemented in parallel form. First, it performs a large amount of non-related, repetitive operations. Additionally, it performs these operations at a large enough scale that the delays in obtaining the output are clearly evident. It is important to note these characteristics because even if an application seems like an excellent candidate for parallel computing, if the end result is not a significant improvement, then the efforts are pointless. In the given scenario of EKG reconstructions, one reconstruction was originally costing them 5

minutes of wall clock time. If the redesigned parallel algorithm reduced the run time to 4.5 minutes then the end result seems hardly worthwhile. However, if the run time is reduced to 1 minute, then it clear that parallelism played a significant role.

1.4 Network Computing

Up to this point parallel computing has been discussed from the point of view of having a process distributed within the cores of a single machine. However, parallel computing is not limited to the resources of a single machine. The underutilized resources of numerous machines within a given network can also be employed in the form of distributed computing [4]. Similar to the way parallel computing was described to take advantage of the resources of multiple cores on an individual machine, distributed computing takes advantages of the multiple cores on multiple machines on a network. By distributing the workload across several machines, the beneficial effects as a result of parallel computing with a single machine are magnified. Also similar to parallel computing on a single machine is the necessity to weigh the costs of implementing a distributed application against the benefits which it will provide.

Initially it may seem obvious that having more machines work on a single problem would benefit rather than hinder the process. However, this is not always the case even when assuming that the problem being addressed meets all the qualifications to be a candidate for parallel computing. A leading reason for this discrepancy is the fact that there is a significant amount of overhead involved in creating the communication between entirely separate systems. Although several methods have been developed to help streamline this process, it is still a more painstaking procedure than coding for a single computer. The overhead created for dealing with the extra communication can add delays that cause run times to increase rather than decrease.

Based on the magnitude of operations required to complete the reconstruction algorithm developed by UTEP's BPG it is evident that it would reap the benefits of being distributed over a network. The simple fact that the QR factorization algorithm needs to be employed 20,571 times provides a good indication that the resources provided by a single machine would be exhausted.

As a result the increased overhead that accompanies distributed computing should not have an adverse effect on the overall run time.

Clearly, distributed computing over a group of machines on a common network can provide huge advantages over working on a single system, but often it is desired to obtain the utmost degree of efficiency. Although, the benefits of network computing are undeniable there is still room from improvement. One source of improvement is addressed by computing clusters. Unlike a network of computers a computer cluster links machines allowing them to work together as if they were one. This subtle difference provides an edge in performance for clusters over computer networks. There are different types of clusters currently in existence including high-availability clusters, load-balancing clusters, grid computing clusters, and Beowulf clusters. Each type of cluster is slightly different depending on its purpose.

The Beowulf cluster is a specific type of cluster that consists of commodity off the shelf components. This approach allows the cost of the cluster to remain low relative to the cost of other clusters and more so than the alternative large supercomputers. Although less expensive, Beowulf clusters are still capable of producing supercomputer performance [5]. The performance of these clusters is so impressive that they have been ranked among the top 125 supercomputers in the world, and they are capable of running on a variety of platforms and used for numerous diverse applications [5, 6]. Given that the Distributed Computing Lab (DCL) at UTEP has recently built and currently maintains an impressive Beowulf cluster, it seems fitting that the reconstruction algorithm developed by the BPG would be parallelized and distributed over the Beowulf cluster to obtain the absolute best speed-up possible.

1.5 Objective

The underlying purpose of this thesis is to work in conjunction with the BPG to help distribute the EKG reconstruction algorithm in an attempt to significantly decrease run times. To achieve this goal, it was first necessary to streamline the current algorithm, by identifying redundant or unnecessary code and creating a more efficient base program. This provided insight on common mistakes made in the implementation of similar algorithms and methods by which they can be avoided. Next, the streamlined program was made parallel in a variety of ways. First, parallel to run on a single multi-core machine; then, parallel to run on a network of machines (which was ultimately run on the Beowulf cluster). Finally, different tests were conducted to identify the benefit of implementing the parallelism via the use of threads as compared to the use of multiple processes. Additionally, this thesis provides the ground work for future implementations of reconstruction algorithms given that this research was conducted during the early stages of the reconstruction algorithm. The end result shows:

1. The speed-up generated by parallelizing this algorithm at the different levels previously described
2. A comparison of the costs and benefits from using threads vs. multiple processes
3. An evaluation of the efficiency depending on the number of nodes included in the cluster

1.6 Organization

The purpose of Chapter 2 will be to provide the theory of the different ideas presented and being worked with. First, it will cover in more detail the methods used in the reconstruction of the EKG signals. Next, it will provide a quick overview of Remote Procedure Calls which is the method by which the distributed computing will be accomplished. This will be followed by a comparison of threads and the forking of new processes. Finally, it will explain how the performance or efficiency of the distributed application is measured. Chapter 3 will describe the application of the proposed ideas. It will begin with an overview of the existing reconstruction algorithm, followed by a description of the changes made to streamline the process and to make the code parallel. Also, this chapter will provide the characteristics of the systems used as well as dependencies of the implemented methods. Lastly, Chapter 4 will review the results.

Chapter 2

Theory

2.1 EKG Reconstruction

The reconstruction of the EKG measured signals is based on a dipole model of the heart. Although both a fixed dipole model and a moving dipole model exist, this research only addresses the reconstruction base on a fixed dipole. This approach allows for a more simple demonstration of the key ideas of this research while still providing a workload sufficient to tax traditional systems. This section will provide the mathematical equations used to justify the approach selected by the BPG in the reconstruction. The group's principal goal is to apply the dipole model of the human heart to a set of precordial leads and verify the model's reconstruction capabilities [2]. The graphical representation provided by the BPG is as follows.

$$\{f\}_N \rightarrow \{v\}_N \rightarrow e = \{f\}_N - \{v\}_N \tag{1}$$

In the above equation $\{f\}_N$ represents the mathematical dipole model applied to N precordial leads. The set of N precordial measurements is represented by $\{v\}_N$, and finally, e is correspondent to error or the difference between $\{f\}_N$ and $\{v\}_N$. This difference is a characterization of the mathematical model's reconstruction capability [2].

Further, the BPG assumes that precordial lead measurements represent time-dependant surface potential measurements that are measured on the body. Therefore, the precordial measurements are represented as the time-dependant function

$$v(t) = f(x, y, z, t) \tag{2}$$

In this function, *x*, *y*, and *z* correspond to the location of the source inside the human body which produces surface potentials as a function of time *t*. However, when the dipole model of the heart is used the function described becomes:

$$f(x, y, z, t) = \vec{c}(x, y, z) \cdot \vec{m}(x, y, z, t) . \tag{3}$$

The spatiotemporal function is represented by $f(x, y, z, t)$, $\vec{c}$ is the spatial lead vector, and $\vec{m}$ is the spatiotemporal dipole moment that acts as the source that generates the surface potentials [2]. Both $\vec{c}$ and $\vec{m}$ depend of the location of the source $\vec{m}$. In order to represent a fixed dipole model it is required that the source be held at a fixed location in space through time. If the location variables *x*, *y*, and *z* are kept constant over time *equation 3* becomes:

$$f(t) = \vec{c} \cdot \vec{m}(t) \tag{4}$$

From the above equation it is clear to see that $\vec{c}$ remains constant while $\vec{m}$ continues to change with respect to time. *Equation 4* is generalized to include multiple surface potential measurements as on a standard EKG. The following shows how the equation is generalized:

$$\begin{array}{ccccccc} v_1(t) & = & f_1(t) & = & \vec{c}_1 & \cdot & \vec{m}(t) \\ v_2(t) & = & f_2(t) & = & \vec{c}_2 & \cdot & \vec{m}(t) \\ \vdots & & \vdots & & \vdots & & \vdots \\ v_N(t) & = & f_N(t) & = & \vec{c}_N & \cdot & \vec{m}(t) \end{array} \tag{5}$$

Since the BPG will be using 6 measurements as part of their reconstruction, the generalized form is utilized. For simplicity this form is re-written as:

$$\{v\}_N = \{f\}_N = \{\vec{c}\}_N \cdot \vec{m} \tag{6}$$

Here, $\{v\}_N$ is a column vector containing *N* surface potential measurements, $\{f\}_N$ is a vector containing *N* mathematical model equations, $\{\vec{c}\}_N$ is a vector containing *N* lead vectors, and $\vec{m}$ is the spatiotemporal dipole source [2]. To illustrate how the reconstruction is

accomplished, which consists of solving for $\vec{m}$, they assume only one surface potential measurement, which means that *N*=1. Therefore, our working equation is:

$$v(t) = \vec{c} \cdot \vec{m}(t) \tag{7}$$

It is important to understand that $\vec{c}$ and $\vec{m}$ are 1x3 and 3x1 vectors respectively [2], such that:

$$\vec{c}(x, y, z) = [c_x \quad c_y \quad c_z] = [c]$$

$$\vec{m}(x, y, z, t) = [m_x \quad m_y \quad m_z]_t^T = [m]_t$$

Hence:

$$v(t) = [c][m]_t \tag{8}$$

As a result of dealing with vectors, or single dimensional matrices, linear algebra techniques must be employed to solve for [*m*]. Although several methods exist to tackle this type of scenario, both QR factorization and the pseudo-inverse methods are used. In order to solve for [*m*] using the QR factorization the Householder reflections are used to compute an orthogonal-triangular factorization [12]. Such that:

$$[c] * P = Q * R \tag{9}$$

Where P is a permutation matrix, Q is orthogonal and R is upper triangular. Next, the solution for [*m*] is computed:

$$[m] = P * (R \setminus Q' * v(t)) \tag{10}$$

The pseudo-inverse approach is determined as follows:

$$[m] = \{[c]^T [c] \}^{-1} [c]^T v(t) \tag{11}$$

After the spatiotemporal dipole source $\vec{m}$ has been computed, the reconstruction or calculation of the mathematical model equations can be performed.

2.1.1 Solution Space

In order to produce useful results the BPG used a Finite Element (FE) model of the human torso obtained from the SCIRun/BioPSE software [2]. This model was used to obtain values to generate a solution space to represent the fixed dipole model of the heart. The SCIRun model was used to generate the values because it simulated realistic internal properties including inner body organs, bones, fat, blood, skin and ribs [2]. The solution space was created using the realistic FE model by assuming possible locations in and around the heart and calculating the respective lead vectors [2]. In all, 20,571 locations were analyzed. Thus, the solution space contains 20,571 sets of values. Each set of values has 6 lead vectors to correspond to precordial leads [2], and each lead vector contains components in rectangular coordinates such that one location would be represented in the solution space as:

$$\begin{aligned}
&\vec{v}(x,y,z)_1 = [v_x \quad v_y \quad v_z]_1 = [v]_1 \\
&\vec{v}(x,y,z)_2 = [v_x \quad v_y \quad v_z]_2 = [v]_2 \\
&\vdots \\
&\vec{v}(x,y,z)_6 = [v_x \quad v_y \quad v_z]_6 = [v]_6
\end{aligned}$$

As a result the solution space consists of a matrix of the size 20,571 x 6 x 3. That is a total of 370,278 values stored as double precision floating point numbers.

Note: The actual size of the solution space matrix is 20,571 x 14 x 3. However, the size 20,571 x 6 x 3 is assumed because these are the values used in the reconstruction algorithm to correspond to the 6 main precordial leads. The discrepancy between the 14 and 6 is as a result of 14 lead locations that can be used in the EKG readings which are not standard.

2.2 Remote Procedure Calls

2.2.1 RPC Introduction

In order to successfully implement a distributed application, some form of communication needs to exist between the parallel threads or processes. This is generally referred to as interprocess communication (IPC), and it encompasses different methods of message passing between different processes that are running [7]. There are several approaches to distribute a process, including distribution within a single host and across multiple hosts; therefore, there also exist different methods to handle communications among the distributed processes. One such method that enables interprocess communication between two or more hosts is referred to as Remote Procedure Calls, or RPC. RPC's are an excellent tool that use implicit network programming [7]. In other words, distributed functions are called just as they would if they were not distributed. However, the process that is calling the function, commonly referred to as the client, and the process that will execute the function, normally referred to as the server, can reside on different hosts. Since the client and server reside on separate machines there has to be some type of network communication involved; however, RPC's take care of handling the communication, and as, a result the programmer is not responsible for it. This transparency allows for a faster implementation of RPC, and although there is some overhead associated with their use, it is negligible compared to the benefits in speed-up that they provide with applications that are parallel to the extent of the reconstruction of the EKG problem.

2.2.2 RPC History

The idea of RPC has been around since the late 1970's and was more formally applied in the early 1980's by Xerox [7]. By the mid 1980's more companies such as Sun Microsystems had understood the importance and impact that the RPC tool would have on the computing

industry, and they released their own version of the RPC package. The official name of the release is ONC/RPC, but it is often referred to as Sun RPC [7]. Another RPC package was also released by Sun which is known as Transport Independent RPC or TI-RPC. The most significant difference between TI-RPC and ONC RPC is the ability for TI-RPC to use different transport layer protocols. Despite the popularity and development of other RPC packages, ONC/RPC remains among the more popular releases [7]. As such, Sun RPC is the package that will be used in the distribution of the EKG reconstruction.

2.2.3 RPC Details

Another advantage of using RPC is that it incorporates the external data representation standard referred to as XDR. This is important because often when distributing applications over a network of machines, the architecture and operating systems of each can differ. These differences translate to different data representations which if unaccounted for would produce erroneous results. However, our implementation of distribution will be conducted on identical machines and thus the data incompatibility problem is not applicable, but it is important to understand that XDR is being used regardless of whether or not the client and servers have compatible data representation. The reason this needs to be considered is because this additional manipulation of the data adds overhead to the implemented RPC distribution.

The first step in building a RPC distributed program is to code the RPC specification file which has a ".x" extension. This file defines the server procedures, their arguments and their results [7]. After the file has been created *rpcgen* is used to generate the necessary files for the RPC implementation. These files include a header file which includes the definitions for the parameters and results as well as function prototypes. Also created by *rpcgen* is a client stub, a server stub, and XDR file that handles the XDR data conversion [7]. The client and server stubs

are responsible for the communication between server and client when the application is running. Next, it is necessary to code the client side program, which includes the calls to the remote procedure, and the server program, that contains the remote procedure. The next step is to compile both of these files with their respective stubs to generate the executables. Finally, the program can be run by starting the server then running the client. From a high level the following steps take place during a remote procedure call.

1. The server starts and sets up its necessary network communication.
2. The client is started, which also sets up network communication, and the client makes the call of the remote procedure.
3. The client stub handles this call and packages the necessary data, or marshals the data, and sends the marshaled data to the server stub.
4. The server stub receives the package and unpacks or unmarshals it.
5. The server stub invokes a local call to have the server execute the procedure with the data received. When the server completes execution it returns the results to the server stub.
6. The server stub marshals the data and sends it back to the client stub.
7. The client stub unmarshals the data and returns it to the client.
8. The client continues to run.

These steps are better illustrated using the *Figure 1* below which shows the flow of control/data using the solid arrows beginning with the client. The dotted arrows illustrate the flow from the point of view of the programmer since all the other steps are virtually transparent to him. Also, it is important to note that the client is blocked after it has made the remote procedure call until the call returns.

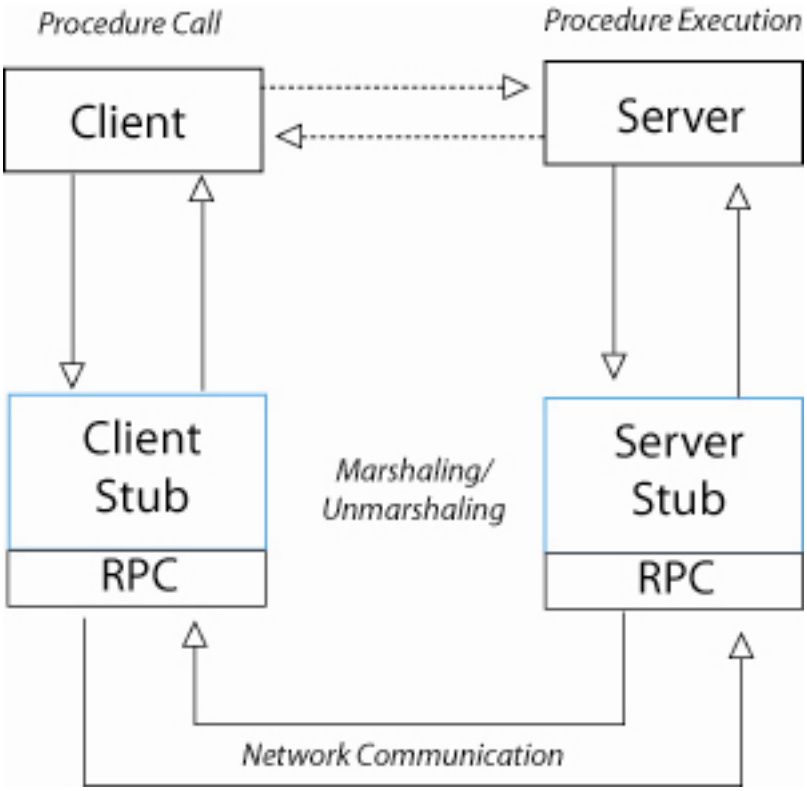

Figure 1: RPC Flow

2.3 Distribution Using Fork

We have now established a basic understanding of what the reconstruction of the EKG consists of. Further, we have justified its qualifications as a good candidate for distributed computing, and we have described the method by which the distributed computing will be accomplished. Now, it is important to understand the means by which the distribution will be employed successfully from the standpoint of the program. This is a paramount aspect of the distribution because we need to take advantage of the available resources. *Figure 2* below shows 3 different scenarios of applying the EKG reconstruction algorithm.

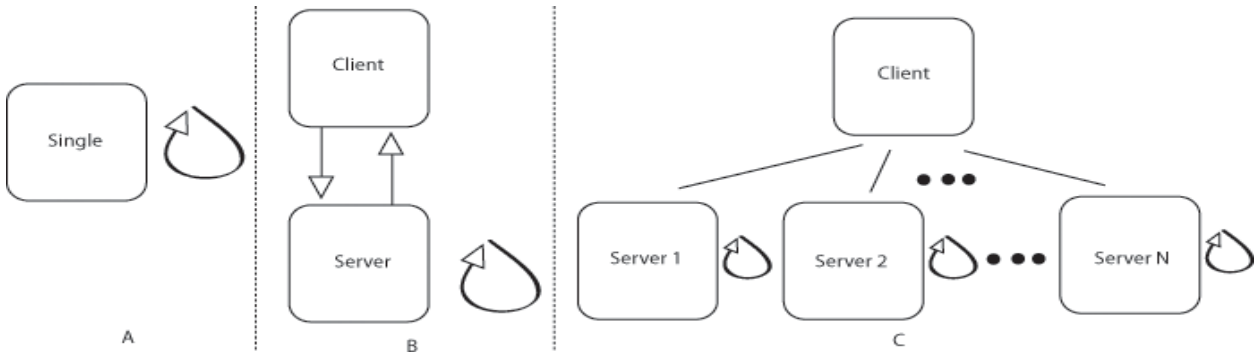

Figure 2: Distributing Reconstruction Calculations

In scenario *A* there is a single machine that is iterating through all the calculations to determine the best EKG reconstruction. Scenario *B* illustrates an attempt to take advantage of the distributed computing by having the client request that a server does the calculations. This would be beneficial if the server is a superior machine and can accomplish the task of reconstruction faster; otherwise, it is an example of how distribution does not always benefit the cause of speed-up. It can certainly cause the calculation to take longer due to the added overhead and delays attributed to the network communication. Finally, scenario *C* illustrates the ideal method to distribute the application. In this scenario the calculations of reconstruction are equally divided between *N* servers. Employing this technique would likely yield the fastest

results, but in order to implement this technique more than one process or thread needs to be running. This is because when a remote procedure is called, the calling process is blocked until the function call returns. Therefore, if there was only one process or the process was single threaded then the maximum distribution possible would be such as that illustrated in scenario *B*, or scenario *C* would occur in serial where the client does distribute the work, but it waits for one server to finish before it can communicate with the next to have it do the work. This would be a less than ideal situation similar to running in serial on a single machine but with the added network communication and RPC overhead.

The first approach would be to use multiple processes. This is similar to the way a UNIX operating system begins with a single process then creates a number of other processes to handle other OS services. To accomplish this, the *fork* system call is invoked at which time a new process is created with identical copies of the user-level context [8]. The *fork* system call will be used in the parallel reconstruction algorithm by having the main process, or parent process, spawn child processes via the *fork* system call. Each child would then be responsible for making the remote procedure call. By this method, only the child processes are blocked by the remote procedure calls, and the parent can continue to make the necessary *N* processes to generate the parallel distributed computing necessary to maximize the efficiency. Under certain circumstances using *fork* is very beneficial; especially in the case where the child process will need to have access to variables and file descriptors that are being used by the parent process. On the other hand, some of the same features that make it attractive can cause unnecessary overhead in the implementations of *fork*. For example, in the case of distributing the reconstruction of the EKG very little information needs to be shared between the parent and the

child processes. Therefore, the procedure of creating a new process and copying the entire user-level context can be more work than necessary.

2.3.1 Shared Memory

The process of providing data by the parent to child is relatively easy; a parent simply needs to set up the data in any type of structure before creating the child, and the data is inherited by default. Sharing data back from the child to the parent becomes a more involved process. This requires another form of inter-process communication such as shared-memory, a technique that allows multiple processes to read from and write to the same memory region. Using this method, processes can communicate directly with each other. As opposed to other methods of inter-process communication shared memory is advantageous because there are no system calls needed to access data in the shared memory region; it is accessible in the same way other memory regions are to a process [8].

2.3.2 Synchronization

In order to use shared-memory, especially in cases such as ours where the shared memory will be written to, some form of synchronization is needed. The synchronization needs to ensure that only one process has access to the shared-memory region at a time. This is done in order to eliminate possible errors or corruption of data that can be caused when more than one process is reading and writing to the same memory region. One such synchronization method is known as a mutex, or a variable that can be in one of two states: unlocked or locked, and it is used to manage mutual exclusion to some shared resource or piece of code [1]. With the use of a mutex we can verify that the values we intend to return to the parent, via the use of the shared memory region, are delivered safely and without interference from the other children also attempting to do the same.

2.4 Distribution Using Multiple Threads

An alternative approach to implement distribution as illustrated in *Figure 2-C* consists of using multi-threading. A thread of execution, or thread, has a program counter that keeps track of which instruction to execute, registers to hold its working variables, and a stack that contains execution history [1]. A thread executes as part of or within a process, but it is a separate entity sometimes described as lightweight-process [1]. Similar to the way that a system can have multiple processes running, a single process can consist of multiple threads executing. Threads are easier/faster to implement than separate processes because they share user-level context including address space, global variables, and open files among the more important. Therefore, threads are intended to work together to accomplish a similar goal whereas one process can fork another to accomplish a completely different and unrelated task. To further illustrate how forking new processes compares to spawning multiple threads refer to *Figure 3.*

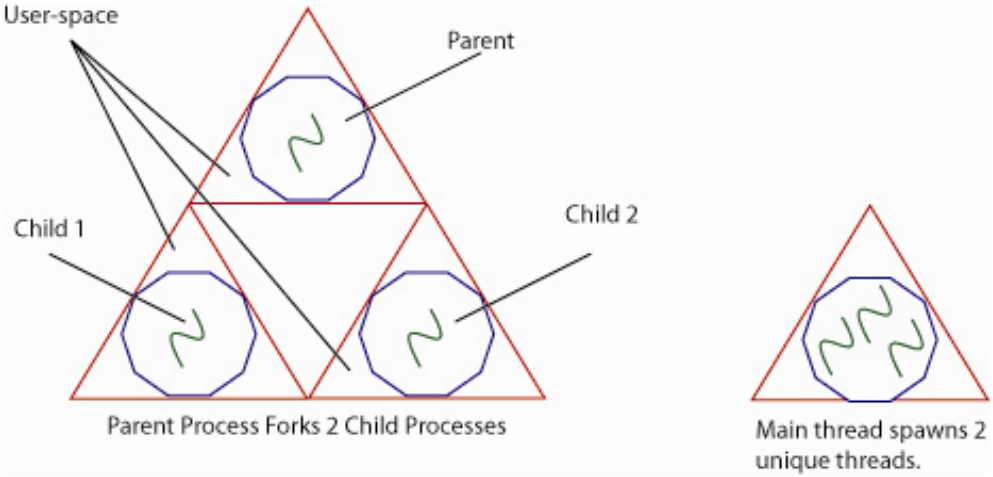

Figure 3: Multiple Processes vs. Multiple Threads

In this image the left hand side shows a parent process after it has created two child processes. The triangle surrounding each process represents the user-space which is duplicated when the process is forked. The user space includes address space, global variables, open files, child processes, pending alarms, signals and signal handlers, and accounting information [1]. The

duplication of this information constitutes the overhead associated with forking a new process. On the right hand side of *Figure 3*, however, the idea of multi-threading is illustrated. From the figure it is easier to see how each thread can operate independently within the same process and as a result share the user-space. The only items that are unique from thread to thread are a program counter, registers, a stack and the state of the thread [1]. Consequently, thread creation should complete faster than the forking of a new process.

In the case of the EKG reconstruction, multi-threading can be implemented in a similar way that multiple processes were used, but instead of having the main process invoke the expensive *fork* system call, it can create alternative threads. Previous work, including research conducted at UTEP, suggests that creating threads is faster than creating a new process [9]. This is a very intuitive conclusion seeing as how a thread shares user space rather than generating a duplicate copy like the *fork* system call does. Each thread, then, would be responsible for calling the remote procedure, and when the blocking occurs as the remote procedure is being executed only a single thread will be blocked instead of the entire process. Under these circumstances, multiple threads can be created at once and the true parallel distribution intended in the reconstruction of the EKG can be attained.

Despite the fact that threads take advantage of shared resources they also pose the same problem described in the use of shared memory. That is, synchronization between the threads is necessary when certain shared resources are accessed. Although shared-memory as described before is not implemented with threads, it occurs implicitly as a result of the nature of their shared resources. Thus, mutexes will also be implemented to protect the variables used to hold values that are returned by the remote procedure reconstruction algorithm. It is expected that

using threads will produce faster computation times over *fork* due to the reduced taxing of the previously described system resources.

2.5 Performance Measure

In order to accurately determine how beneficial the distribution of the EKG reconstruction has been there has to be some type of performance measure. The first way that the benefits can be determined is by figuring out how much faster the distributed applications are running. This measure is called the speed-up; the calculation for which is commonly known to be as follows [11]:

$$S_p = \frac{T_1}{T_p} \tag{2.5-1}$$

Where S_p refers to the speed-up resulting from *p* number of processors. Further, T_1 is the execution time for the original, or serial, algorithm, and T_p represents the execution time of the parallel algorithm utilizing *p* processors. Ideally, speed-up would increase linearly with respect to the amount of processors used such that with 2 processors the program would run twice as fast, with 3 processors the program would thrice as fast and so on. However, Amdahl's law dictates that this is not possible unless 100% of the program being made parallel can be parallelized, which is rarely the case [11].

Another measure of performance that will be evaluated will be that of efficiency. This measure is important because it dictates how well the resources are being used. The formula commonly used to calculate the efficiency is as follows [11]:

$$E_p = \frac{S_p}{p} = \frac{T_1}{\left(p * T_p\right)} \tag{2.5-2}$$

Here E_p is indicative of the efficiency while S_p continues to represent the speed-up, and *p* shows the number of processors being used. The efficiency is also shown as a function of the execution times on the far right side of the equation. From the equation to calculate efficiency it is clearly seen that an ideal efficiency of 1 would be obtained if linear speed-up was attained. Since we know that linear speed-up will not occur, then we expect our efficiency to be within a range of 0 and 1.

In analyzing the parallelization of the EKG reconstruction algorithm both the speed-up and efficiency will be accounted for. The speed-up will provide a sense of the benefits obtained from the distribution, and the efficiency will dictate how many servers to use to obtain the best balance in speed-up and use of resources. It is important to understand that although adding more servers may continue to increase the speed-up, it may not be worth the effort if the speed-up is not sufficiently large. This will be determined by the calculation of efficiency.

Chapter 3

Application

3.1 Original Algorithm

Before being able to describe how an application will be distributed in parallel, a thorough understanding of such an application is required. The same is true for the algorithm used by the BPG to reconstruct an EKG. For starters, this algorithm was implemented using Matlab, the high-performance language for technical computing. Matlab is great tool that provides an enormous diversity of assistance for a range of applications including the following described on the product website [12]:

- Math and computation
- Algorithm development
- Data acquisition
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

With such a diversity of tools accessible within Matlab it is very easy to implement algorithms including that of reconstructing an EKG. However due to its size, Matlab also consumes more resources than a more direct or focused application would. Despite the ease with which the BPG was able to implement their algorithm using Matlab, they also experienced the repercussions of the large application including long delays to obtain their results. Subsequent sections of this

chapter will be dedicated to describing how this issue was addressed, but first, a description of the Matlab programs/scripts utilized will be given.

3.1.1 Main Function

The bulk of the responsibility of the reconstruction was assigned to a function appropriately named *obtainbestpoint* [2]. This function requires the following input parameters:

- **leadvector** - Used to pass the lead vector (or solution space as previously described) that will be used as a fixed dipole model of the heart in the EKG reconstruction. Represented in Matlab by a matrix of double precision floating-point values with dimensions of 20,571 x 6 x 3.
- **ECGoriginal** - This parameter contains the values for a measured EKG of a patient. Used as the model to be reconstructed and as a result used to calculate the accuracy of the reconstructed signal. Represented by a matrix of double values of size 12 x 302.
- **ind0** - Used to control the starting index that will be used from the **leadvector** in the reconstruction algorithm.
- **ind1** - Used to control the ending index that will be used from the **leadvector** in the reconstruction algorithm.

Upon entering, the function begins a loop controlled by a variable beginning with **ind0** and ending with **ind1**. For each iteration of the loop, a set of values corresponding to the loop's control variable is obtained from **leadvector**. This set of values is known as the spatial lead vector, or just lead vector, in the theoretical description of Chapter 2. First, the rank of the lead vector is obtained, using the Matlab *rank* function, and stored in an array. The lead vector, along with the original EKG values, is then used to calculate the dipole moment. This calculation is attained by using Matlab's *mldivide* function. For now it is sufficient to say that it employs QR

factorization to arrive at the correct solution. Once the dipole moment is obtained, it is used in conjunction with the lead vector to reconstruct an EKG signal. The new EKG is then fed into an error calculating function, which will be described later, to determine the root-mean-square (RMS) error and store the error in an array indexed by the current value of the main loop's control variable. This process is repeated until all iterations of the loop have completed. Upon completion the function returns the following values:

- **MinError** – The error value for the index which produced the smallest error.
- **Index** – The index of the lead vector that produced the smallest error. This value along with the values that it indexes is considered the solution for the reconstruction.
- **sumerror –** The array containing the error values for all reconstructions. Each error value is associated to the lead vector by the index.
- **rankvector –** The array containing the rank of each lead vector used in the reconstruction.

The function also displays its run time in microseconds. This value is later used to determine the speed-up obtained by the distributed computation.

3.1.2 Error Functions

Next, a description of the functions used to calculate the error is given. The first of the two functions is called *NRSM12*, which consists of a loop that iterates 12 times. Although no error computations occur within this function, it is used to send values to the function (*NormRMSError*) that does the error computations. The inputs of *NRSM12* are the following:

- **EKGorig** - This parameter contains the values for a measured EKG of a patient. Matrix of double values of size 12 x 302.

- **EKGcompared** – This parameter contains the values that were computed as a reconstruction attempt. Matrix of double values of size 6 x 302.

Every cycle of the loop within this function is used to send one of the 12 sets of values, or vectors, from **EKGoriginal** and **EKGcompared**. Each vector consists of 302 values to correspond with every time sample of the measurement. The values returned from the error calculating function (a vector) are stored in a matrix which is the single output value of *NRSM12*.

Note: The EKG reconstruction algorithm employs only 6 precordial leads, but the original EKG values consist of all 12 possible leads. Therefore, a padding of zero values is added to have the compared or reconstructed EKG match the size of the original EKG.

The function *NormRMSError* is employed to handle the error calculation between the reconstructed and original EKG. Its input parameters consist of two vectors; one vector from the original EKG and the second from the reconstructed EKG. These parameters are:

- **Xact** – Contains the vector from the original EKG.
- **Approx** – Contains the vector from the reconstructed EKG.

First the function normalizes the values within each vector using the maximum absolute value from **Xact.** The normalization is accomplished by using the Matlab functions *max* and *abs*, which find the maximum within a vector and the absolute value respectively. Next, the function enters a loop that will be performed 302 times, or the number of values contained within the vectors. For each cycle the difference between the indexed value of **Xact** and **Approx** is calculated and stored. After the loop completes the RMS is calculated by taking the square root

of the average of all the values squared, or differences squared. The RMS is then returned as the single output of the *NormRMSError* function.

3.2 System Description

To provide a fair comparison, the distributed version of the EKG reconstruction and the original algorithm need to be executed on comparable hardware. Otherwise, any speed-up and efficiency measurements obtained can simply be attributed to the superiority of one system over the other. In the case of the Beowulf cluster being used to distribute the reconstruction vs. the machines used by the BPG to process the original algorithm in Matlab, the discrepancy between systems is enormous. As a result, all version of the EKG reconstruction code including the original Matlab algorithm and the various redesigns attributed to the distribution procedure are run on the new Beowulf cluster recently built by the DCL group at UTEP known as Virgo 2.

3.2.1 Cluster Attributes

Virgo 2 is an advanced Beowulf cluster built with funding provided by the National Science Foundation. It consists of a frontend, 21 compute nodes, and 2 memory nodes. However, for the purposes of the distributed reconstruction the memory nodes are treated like compute nodes because the extra memory does not affect the computation times of the reconstruction. More specifically the front end consists of:

- Tyan Tempest i5400 Series Motherboard
- Two Intel's Harpertown Quad-core Xeon processors
- Kingston 8GB FB-DIMM DDR2 RAM
- Four 500GB (16MB cache) SATA2 Seagate Hard-drives
- One 250GB (16MB cache) SATA2 Seagate Hard-drive
- Single Channel SCSI Card
- PNY Quadro FX1700 Graphics Card
- LITE-ON DVD ROM

Additionally, each compute node consists of:

- Tyan Tempest i5400 Series Motherboard
- Two Intel's Harpertown Quad-core Xeon processors
- Kingston 8GB FB-DIMM DDR2 RAM
- One 250GB (16MB cache) SATA2 Seagate Hard-drive
- Gigabyte 128MB GeForce 7200GS Graphic Card
- LITE-ON DVD ROM

Lastly, the memory nodes consist of the following hardware:

- Two Intel's Harpertown Quad-core Xeon processors
- Tyan Tempest i5400 Series Motherboard
- Kingston 64GB FB-DIMM DDR2 RAM
- One 250GB (16MB cache) SATA2 Seagate Hard-drive
- LITE-ON DVD ROM

Additionally, Virgo 2 is running the CentOS release 5 as part of the open-source LINUX distribution of ROCKS version 5 to manage and monitor the system. The cluster also depends on a dedicated network for communication among the nodes. To handle the network communication 3 Cisco 3750 24 port gigabit Ethernet switches are used. The current fine-tuning of the Virgo 2 cluster has resulted in performance speeds exceeding 700 TFLOPS. However, more fine-tuning is expected to increase the performance.

3.3 Dependencies

3.3.1 BLAS

In order to avoid the extra overhead generated by Matlab in the implementation of the EKG reconstruction, the distributed routine is developed in the C programming language. However, the standard C libraries do not include functions that are capable of performing advanced mathematical computations such as the ones necessary in the EKG reconstruction. As a result other packages need to be installed, and as such are defined as dependencies for the applied distributed program. One of these packages includes the Basic Linear Algebra Subprograms, otherwise referred to as BLAS. The BLAS, as the name suggests, are a group of functions that can be used in the computations of basic vector and matrix operations [10]. BLAS is subdivided into three levels, each having a higher degree of complexity. The breakdown of the levels is as follows:

- Level 1 BLAS – Provides routines to perform operations dealing with scalars, vectors, and vectors with vectors.
- Level 2 BLAS – Provides routines to perform operations dealing with matrices and vectors.
- Level 3 BLAS – Provides routines to perform matrix-matrix operations.

Due to the nature of the reconstruction algorithm, it calls for the use of Levels 2 and 3 of BLAS. These are necessary mainly when applying the QR factorization to obtain the dipole moment for a lead vector, but this process will be described in further detail later. Since the BLAS routines have proven to be so useful as a result of their efficiency, portability and availability, they are used to create linear algebra software [10]. The BLAS subroutines are not called directly, instead they are called via the package known as the Linear Algebra Package, or LAPACK.

Another advantage of BLAS is that different revisions of its libraries are available to provide higher optimization for a variety of architectures allowing for reduced execution speeds [10].

3.3.2 LAPACK

The Linear Algebra PACKage known as LAPACK was originally written in FORTRAN [10]. This package provides routines to handle a variety of tasks common in linear algebra including:

- Solving systems of simultaneous linear equations
- Least-squares solutions of linear systems of equations
- Eigenvalue problems
- Singular value problems
- Matrix Factorization including LU, Cholesky, QR, SVD, and Schur.

One of the goals of the reconstruction distribution is to provide solutions that are equivalent to those currently being generated by the BPG algorithms. Therefore, in the distributed reconstruction the QR factorization needs to be employed similarly to the way that it is used by Matlab. It is in the implementation of the QR factorization into the C code used for the distribution that the LAPACK routines become necessary. Since LAPACK is written in FORTRAN, it is not convenient to directly call its functions from a C program; yet, the popularity and use of C have grown drastically. Consequently, the C version of LAPACK known as CLAPACK was built using a Fortran to C conversion tool called *f2c* [10]. Using CLAPACK it was possible to implement the QR factorization necessary in the reconstruction of the EKG. Although both CLAPACK and BLAS are dependencies of the distributed reconstruction, it was only necessary to install CLAPACK because the release includes a generic version of the BLAS libraries that are referenced from the routines used in the QR factorization.

3.4 Data Conversion

Certain data elements used in the original Matlab algorithm were also necessary in the C version of the algorithm. In particular the solution space containing the lead vectors and the original EKG data that represents patients were required. These data elements were contained in a Matlab workspace and formatted specifically for Matlab. For this reason, the data elements needed to be exported out of Matlab into a file that could be read by a C program. To address this issue a conversion function was written in Matlab. Although the solution space remains constant for all reconstruction attempts and would only be required to be converted once, different original EKG patient data may need to be reconstructed. Hence, a conversion function that was general and could address multiple patient data sets was required. In order to ensure that the output generated by the conversion function would be compatible to the C version of the reconstruction, the conversion function was written in C. In order to be compatible with Matlab a special C file was coded called *convert.c* that contained a *mexFunction* which becomes the entry point of the code rather than the standard *main* function. When the C file was compiled using Matlab's *mex* command Matlab generated a MEX-file with the name of the C file; in this case it was called *convert*. This special handling of the C file was necessary to be able to call the C function from Matlab, but more importantly to be able to pass Matlab workspace variable into the memory space of the C program. In the end the *convert* function that was implemented required two arguments. The first argument was the data structure to be converted and the second argument was a filename in which to save the converted data. The following is an example of how the *convert* function would be invoked to convert the data contained in a Matlab variable called **NormalPatient** and saved to a file called normalpatient.d:

>>convert(NormalPatient, 'normalpatient.d')

After this function has been run, a regular C program could be written to read the data from normalpatient.d and use it in calculations. A key feature of this function is that it can handle variables with up to 3 dimensions automatically, and it contains various means of error checking to ensure the converted files are generated correctly.

The following steps summarize what is involved in the execution of the *convert* program:

1. Verify that two arguments are received. If not, then display error message and exit.

2. Verify that the Matlab call is not expecting any returned values. Otherwise, display error message and exit.

3. Verify that the second argument is a valid string to generate a file by that name. If not, then display error message and exit.

4. Obtain the size of the string, and dynamically allocate enough memory to hold the string and a null character in a variable: **filename**.

5. Read in the second argument and store in **filename**. If an error occurs display error message and exit.

6. Determine how many dimensions the first argument has, and allocate memory for an array of integers to hold the values of each dimension. Store the values of dimensions in the array.

7. Create and open a file name **filename** for writing. If an error occurs display error message and exit.

8. Allocate enough memory and store the first argument into **matrix**.

9. Depending on the number of dimension of the first variable one of two nested loops is performed that writes the values contained in **matrix** to the file using the *write* system call. Each *write* system call has error checking.

It is also important to mention that the standard C header files *stdio.h* and *fcntl.h* are included for standard procedures and function calls. Additionally, *mex.h*, a header file created to be used with Matlab compatible C code is used. The following functions used in the *convert.c* source code are declared in *mex.h*:

- *mxErrMsgTxt* – This function was used to display error messages. It also terminates execution when called.
- *mxGetM* – This function was used to obtain the value of a dimension. Specifically, the number of rows in a parameter.
- *mxGetN* – Also used to obtain the value of a dimension. More specifically, the number of columns in a parameter.
- *mxIsChar* – This function was used to verify that the second argument consisted of characters.
- *mxCalloc* – Used to dynamically allocate memory. Matlab's version of *calloc*.
- *mxGetString* – Used to retrieve the string that was the second argument.
- *mxGetNumberofDimensions* – Used to determine the number of dimensions of the first argument.

- *mxGetDimensions* – Used to obtain the specific dimensions of the first argument.

- *mxGetElementSize* – Used to obtain the size of the elements in the first argument in order to know how much memory to dynamically allocate.

- *mxGetNumberOfElements*- Used to obtain the number of elements in the first argument. Also used to determine how much memory to dynamically allocate.

- *mxGetData* – Used to retrieve data from the first argument.

Another important aspect of the conversion code has to do with how the data in the files is being stored. Typically, in C, matrices are stored in memory in row-major form. This means that all the values in one row are stored consecutively followed by the values of subsequent rows. However, Matlab takes a different approach at storing matrix values in memory; it uses the column-major form. In column-major form, the values of a column are stored consecutively followed by subsequent columns. *Figure 4* below illustrates the difference between column-major and row major matrix data storing. It depicts how a 2 x 3 matrix shown in the upper center of the image is stored in memory in both C, on the left, and Matlab, on the right. Additionally, it shows how matrices can be corrupted if the conversion between the two storing methods is not handled correctly. The bottom of the picture shows

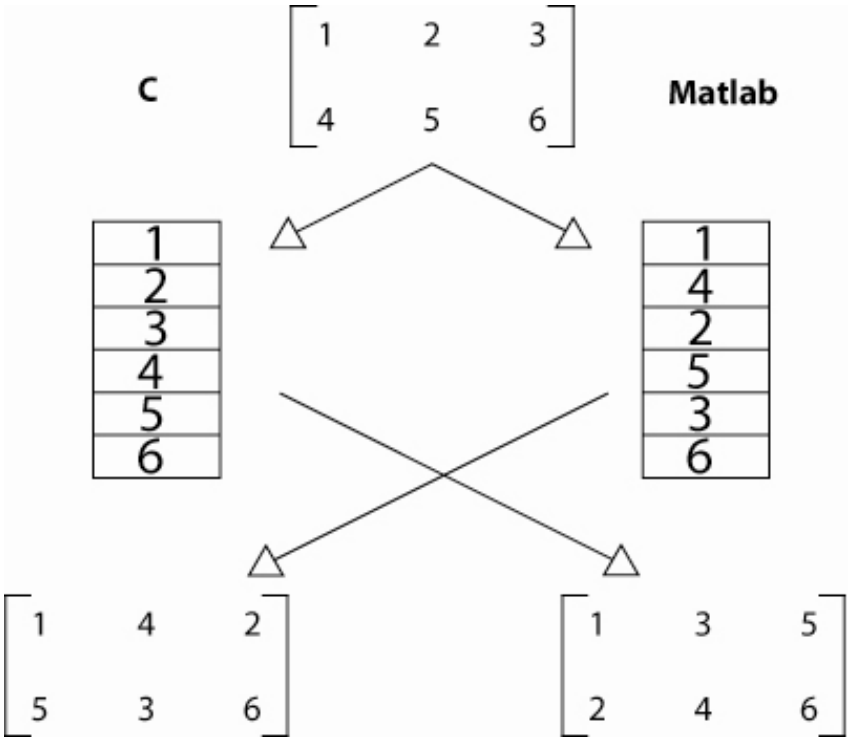

Figure 4: Column-major vs. Row-major Matrix Data Storage

the kind of matrix would be created if data stored in the opposite form is read directly into it without any conversion. Clearly, the newly created matrix does not match what it was initially intended to be; this type of data discrepancy would cause erroneous results. FORTRAN happens to store matrices in memory in column-major form also. Therefore, the CLAPACK routines that are called, despite being in C, still expect the matrices in column-major form. As a result, the conversion program was not written to convert from column-major to row-major form. The main conversion taking place is that of the data representation specifically in how variable types are stored. Despite the compatibility between Matlab and the CLAPACK functions that will be used for the reconstruction, care still needs to be taken to handle the data correctly when it is traversing through the C code.

3.5 Serial Redesign

The first step in the process of distributing the EKG reconstruction was to rewrite the Matlab functions used by the BPG as C programs. This conversion was done to side-step the overhead related to Matlab that was previously described. Further, the purpose was to generate a very specific and targeted program that would strictly be dedicated to the problem of EKG reconstruction. Despite the overall goal to create a distributed application for the reconstruction, the first iteration of the redesign was a sequential implementation. By approaching the distribution in steps, rather than attempting to generate the distributed code all at once, a better understanding of the final results can be attained. For instance, in the serial redesign several inefficiencies of the original Matlab functions are addressed. Comparing the run times of the original algorithm and the serial C code will give us an understanding of the benefit obtained from addressing the suspected inefficiencies, as well as the reduced overhead from eliminating the use of Matlab. In a similar way, comparing the distributed code against the serial code focuses the analysis of the speed-up attained strictly on the distribution.

3.5.1 Inefficiencies

The majority of the inefficiencies addressed from the original algorithm consisted of unnecessary or duplicate calculations. In other words, certain values were computed multiple times, but there was never any possibility for those values to change. The bulk of these redundant computations were done in the error calculation. First of all, when attempting the reconstruction for a particular patient, that patient's original EKG values will remain the same throughout the run. However, the original algorithm sent the original patient data to the error calculations every one of the 20,571 iterations. Meaning that within the error calculations the vectors corresponding to the original EKG were normalized redundantly. As a result of the steps

involved in the normalization procedure, this redundancy wasted a lot of system time. To recap, the original EKG data vector, referred to as **Xact**, is normalized by first obtaining the absolute value of all 302 elements which it contains. Then, these values are parsed through to obtain the maximum value. Finally, every single element within this vector is divided by that absolute maximum value, which produces 302 divisions. Because each original EKG value contains 12 of these vectors the number of divisions increases to 3,624. However, due to the fact that this same error computation routine is initiated for every lead vector in the solution space, the number of divisions calculated for every reconstruction jumps to 74,549,304. In reality these vectors should only be normalized once; meaning that only 3,624 divisions need to occur. Since that is not the case then each reconstruction attempt is producing 20,571 times the number of necessary division computations, as well as absolute value computations, and parsing through a vector of 302 elements. Instead, a more appropriate approach would have been to find the maximum absolute value for each vector within the original EKG values once and store them in a separate data structure. Also, normalize the values and store them in their dedicated data-structure. Then for each error calculation, instead of passing the original values, the normalized values would be passed as well as the max-abs value data-structure. In this manner only the reconstructed data is normalized every time, and the previously normalized original data is readily accessible for the rest of the error computations.

Another source of wasted resources within the error calculations comes from padding the reconstructed data with zeros in order to match the size of the data-structures which contain the original data. This technique is taking a matrix that is 6 x 302 double values, and converting it to a matrix twice the size at 12 x 302 double values. It wastes resources because it adds computations to the error calculation for which the result is already known. For example, in the

process of normalization when the zero values are being divided by the max-abs value from the original EKG, we know that those results are going to be zero. Furthermore, when the difference is being calculated between the normalized original values and the normalized reconstructed values, it is known that the difference for the zero padded elements is going to be the value of the normalized original value. To avoid these unnecessary computations code could be added to skip the calculations for the known range of values that is padded with zeros. Another, more efficient approach, would be not to pad the reconstructed values with zeros but to shrink the original values to match the size of the reconstructed value matrix. This approach is legitimate because excluding the set of values from the original EKG that would coincide with the region of padded zeros does not affect the final outcome of the error calculation. Also, it is more efficient because aside from deducting 1,812 divisions necessary from the normalization routine it also reduces the number of subtraction calculations necessary by the same amount. All in all, it is crucial to beware of the possibility of redundant computations especially within nested loops, otherwise abundant resources will be wasted.

3.5.2 Algorithm

Aside from addressing the inefficiencies mentioned from the original reconstruction algorithm, the serial algorithm in C is very similar to the original. The C code, like the Matlab function, accepts command line arguments to determine the starting and ending index for the main loop that will read through the solution space. Then, still mirroring the Matlab function, it iterates through the main loop each time using a different lead vector from the solution space to calculate the dipole moment, then to reconstruct the EKG and finally calculates the error of that lead vector reconstruction. It the end, the C code also returns the minimum error value and the

index associated with that minimum error which is considered the solution or closest reconstruction point.

Despite the similarities in an overview of the algorithms, the implementation produces a lot of differences. Most of the differences are a result of having to manually code a large amount of the activity that occurs in the background with Matlab. The Matlab function used in the reconstruction algorithm consisted of a mere 53 lines of code, while the C code used to implement the same algorithm exceeded 233 lines. As a result, one may be lead to believe that using Matlab is a far better approach, but in the long run, when obtaining the results, it is clear that the manual implementation is far superior. For example, one major difference is how each implementation obtains the required data. Matlab allows you to save a workspace, which can contain a number of variables, files, and file descriptors. By simply opening the workspace the user has access to all the data contained within, which is automatically arranged and contained in easily accessible data structures. However, while working in C the programmer must open and read the file, and parse the data of the file to organize it into the required data structures manually. The tasks that Matlab handles automatically extend much further than data organization. For instance, to calculate the dipole moment in the Matlab function a single line of code was used:

m=c\ECGoriginal(7:12,:);

This line of code prompts Matlab to automatically determine the types of data involved in the calculation. For example, it determines whether vectors or matrices are being used, and it determines if the matrices are square or rectangular as well as other points about the data. From the information it gathers it then decides the best approach to arrive at the correct answer. It is sufficient to say that it chooses from up to ten different methods of reaching a solution. In the

case of coding the algorithm in C the user must decide how to implement the computation. Because the computation will be repetitive and the data will be similarly formatted during different runs, it makes sense, in this case, to generate a program that will always compute the dipole moment using the same approach. As a result, the code generated in C is more efficient in implementing the computation because it does not need to waste time trying to determine which algorithm to utilize. On the other hand the Matlab form is more versatile in that it can handle a wider range of problems. Again, for the problem at hand of EKG reconstruction, the versatility of Matlab is causing slowdowns in the computation while the C code is fast and direct. The following steps summarize the process of the serial reconstruction of an EKG written in C, *Reconstruct.c*.

1. Variables are declared and space is dynamically allocated for matrices that will be accessed via pointers according to their size.
2. The Starting and Ending Index are set from command line arguments, with error checking.
3. The data files are opened and read into memory. Pointers are set to access data.
4. The array that contains the maximum absolute values of the original patient vectors, **MaxAbs**, is initialized to zeros. This vector will later be used in a condition statement when normalizing the original patient data.
5. The main loop iterates with a control variable beginning with the starting index and finishing with the ending index provided at the command line.

   Steps 5.1 and 5.2 are done to retain original data read in from files. Because variables used in subsequent calls to the CLAPACK routines change are overwritten.

5.1. The original patient data, **pat**, is copied to a memory region pointed to by a different pointer **B.**

5.2. The lead vector data, **mat**, is copied to two separate memory regions pointed to by distinct pointers **A** and **C**.

The following calls to CLAPACK routines are used to obtain the QR factorization of the lead vector to solve for the dipole moment as expressed in equations 9 and 10.

5.3. The CLAPACK routine *dgep3* is called to obtain the QR factorization with column pivoting of matrix **A** [10], or the current lead vector.

5.4. The values in the pivot vector, **JPVT**, are used to generate a pivot matrix, **pivot**.

5.5. The CLAPACK routine *dormqr* is called to multiply the orthogonal matrix **A** generated by the *dgep3* routine with the patient data matrix **B**.

5.6. The result matrix **B** from the *dormqr* routine is transferred from its current size matrix of 6 x 302 to its new size 3 x 302 in matrix **M**.

5.7. The CLAPACK routine *dtrtrs* is called to solve the triangular system using the triangular matrix **A** that was generated by the *dgep3* routine and the product returned by the *dormqr* **M.**

5.8. The CLAPACK routine *dgemm* is called to multiply the output matrix **M** generated by *dtrtrs* with the pivot matrix to align columns appropriately and generate dipole moment matrix **L**.

5.9. The CLAPACK routine *dgemm* is called once again to create the reconstructed EKG signal, **NEW,** by multiplying the dipole moment, **L**, against the current lead vector, **C**.

5.10. If the **MaxAbs** array is still initialized to zero, then the maximum absolute value for each vector of the normal patient array is found and stored in **MaxAbs** array. Then the

normalized version of the normal patient array is created, **NormXact**. Checking that the **MaxAbs** array is initialized before performing this step ensures that the same mistake implemented as part of the original algorithm where the normalization processes was unnecessarily repeated does not occur.

5.11. The RMS error, **RMSerr**, is calculated using the normalized patient vector, **NormXact**, along with the reconstructed vector, **NEW**, and the sum of the RMS error values is stored in **SumErr**.

5.12. The minimum error variable **MinErr** and the minimum error index **MinInd** are populated with the current index and the current error if the current error, **SumErr,** is less than the minimum error. In other words when the current **SumErr** is less than **MinErr**.

6. Display results.

This program also uses the *gettimeofday* function to calculate the run time in microseconds. The run time is also displayed at the end along with results. From the summary it is clear that the implementation in C is far more involved than the functions used within Matlab, but, as mentioned before, the extra work required in the coding reaps the benefits of generating results more quickly. The extent of the improvement in speed will be shown in the next chapter.

3.6 Distribution Using Fork

3.6.1 RPC Specification file

In order to create the distributed version of the reconstruction algorithm which employed RPC's the first step was to generate an RPC specification file. These types of specification files provide server procedures along with their arguments and results [7]. The specification file created consisted of a struct called **rec_in** that was used to contain the arguments to the remote procedure and another struct called **rec_out** which contained the outputs of the remote procedure. The specification also contained an RPC program named **RECONSTRUCT** that consisted of one version named **REC_VERS**. That version contained a single procedure named **qrerror** which had one argument of type **rec_in** and returned a result of the type **rec_out**. To compile the specification file the *rpcgen* function was used, and it generated the necessary files needed to employ the RPC, including the client and server stubs and a header file. Next it was necessary to create the code that would be used on the client machine and the code that would be executing on the server machines.

3.6.2 Client

Having a serial version of the EKG reconstruction in C provided a starting point to use in the distributed code. First the client code was created, *client.c*, which was responsible for determining how many servers where going to be used and separating the workload evenly among those servers. To accomplish this task the client accepted server names as command line arguments. Depending on the number of arguments the client would then use the *fork* function to create that number of child processes. Because the number of servers to use could vary from one run to the next, a loop was used to issue calls to the *fork* function. This loop would cycle through the same number of times as the number of servers, and in each iteration issue a call to

fork. The returning child process id was stored in an array, and the child determined how many indices it was responsible for. Additionally, each child would call the *clnt_create* function to create a client-server connection that would be used in the calling of the remote procedure. Next, each child would call the actual remote procedure function *qrerror*. While the remote procedure was being executed the child is blocked, but after its return the child was responsible for obtaining a mutex lock and updating the minimum error and minimum index variables if applicable. In other words, if the remote procedure call produced the smallest reconstruction error in comparison with the current smallest error it would then become the current smallest error. As a result of generating child process to handle the reconstruction, it was also necessary for the parent process to wait for each child process to finish. This was accomplished by issuing a call to the *waitpid* function for each child that was created, and it ensured that the parent process would not continue until all of its child processes had completed.

Another key aspect of the client code is its implementation of shared memory. As previously described each child would compare the error obtained from the call to the remote procedure with the minimum error. Without the use of shared memory each child process would have had its own minimum error, and as a result, the return from its remote procedure would have always been the smallest error. However, because shared memory was incorporated all processes had access to the same region in memory that was storing the minimum error. Therefore, each child process was able to compare its error with the error for all other process to determine which was the smallest. As a result of using shared memory, it was also possible for the parent to have access to the same region in memory to display the smallest error and associated index for the reconstruction.

3.6.3 Server

The general approach in the use of RPC's is to have the client distribute a bulk of the work to the servers. Therefore, the code for the server is more involved than that of the client, and it more closely resembles the code used in the serial approach. The main exception being that control variable for the main loop is not dependant on command line arguments but rather the arguments used as inputs when calling the remote procedure. Also it is important to note that unlike the serial approach, the distributed approach always iterates through the entire solution space. In addition, the server function returns the smallest error that it computed along with its associated index as part of a struct. Finally, the server incorporates the use of static variable used as a Boolean variable to determine if the files from which it reads the lead vector data and the original patient data have been opened. Through the use of this Boolean variable the server is able to open the files only once and keep their contents in memory. Therefore, once a server is started, it can remain active across multiple calls from a client and it will not be required to open the files each time. As a result, valuable system time is saved by not requiring calls to the *open* and *read* functions for every reconstruction attempt.

3.7 Distribution Using Threads

The use of the *fork* system call was required to implement the distributed reconstruction in order to avoid the RPC from blocking the parent process. Otherwise, if the parent process was blocked, then full parallelization of the reconstruction would not have been possible. However, the *fork* system call can be described as an expensive function with regards to its consumption of system resources. Therefore, to further decrease the computation times required in the reconstruction of the EKG signals, a version of the distributed code was generated that incorporated multi-threading. With the use of multi-threading, the use of *fork* to create child processes was no longer required, and as a result, the copying of the user-context data for each server that is used in the reconstruction was avoided. Although threads will be implemented in a similar fashion that child processes were used, several subtle differences exist between the two. The most prevalent difference has to do with the memory space. All threads within a process share the same memory space; therefore, care needs to be taken to reserve memory regions for specific threads. Otherwise interference can occur among the threads causing erroneous results. To further explain how this can be an issue we will consider how it applies to the distribution of the EKG reconstruction and present the manner in which it was handled. Similar to the *fork* implementation, the thread implementation dedicated each new thread to call the remote procedure. When calling the remote procedure, a starting index and an ending index is required to pass as an argument. Further, the remote procedure returns an error value and an index to the calling process or in this case the calling thread. With the *fork* implementation the parent process simply declared variables to store the starting and ending index and the error and index returned from the procedure call; when the parent process forked a child all those variables were in the unique memory space for each process. Therefore there is no possibility for interference

among the processes. However, if this approach was taken with threads then several possibilities of interference would exist. First, when the starting and ending index are set for one thread if the thread does not call the remote procedure before the same index values are set for the next thread then the possibility exists that two or more calls would be made with the same values. Also when values are returned by the remote procedure, if one thread returns before a previous thread can compare its returned values to the minimum error and index, then those values would be lost. Similar possibilities of interference exist with other variables. To address this issue a struct was defined to hold the following variables:

- **cl** – a client handle.
- **host** – A string of the host name.
- input – A rec_in struct which contains the arguments for the remote procedure.
- output – A rec_out struct which will hold the results returned by a remote procedure.

An array of these types of structs was then created, and each new thread was assigned a different struct of that array. With this setup each thread would only access the variables assigned to it; thus, the possible interference was avoided. Another difference between the thread and fork implementation relates to the fact that when creating a thread a function needs to be specified for the thread to execute. Therefore, the task of creating a client handle and calling the remote procedure was placed in this newly created task. These differences were implemented into the source code of the client side. The server side source code remained the same as that used in the fork implementation.

3.8 Multiple Cores

As mentioned in the second subsection of this chapter, the Beowulf cluster used in the implementation of the distributed algorithm consists of multiple nodes with two multi-core processors. In order to better, or more fully, utilize the computing resources available in each node another version of the distributed algorithm was created. This new version takes the threading implementation to a new level by employing the use of threads not only in the client side, but also on the server side. The goal of this approach is to have 8 threads running in parallel in each of the servers thereby maximizing the use of all four cores in each of the processor on each compute node. To accomplish this, the client code remains the same as described in the initial threaded implementation, but the server code is modified to create the threads. Similar to the changes necessary to client code, in the server code several of the variables that need to be unique are placed into a struct. Also, the main body of the computations, which includes all the calls to CLAPACK subroutines, and the error calculation are moved into the function which will be designated to the threads. Additionally, logic was added to determine how much of the computation responsibility within the given node would be assigned to each thread. As a result the small subsection of the solution space that was assigned to a compute node is then further subdivided into smaller assignments for each thread. Another feature that was necessary was to protect the code that made calls to the CLAPACK subroutines using a mutex. In other words, the process of obtaining the QR factorization was made atomic. Originally, the processing of the solution space on the server side was done in serial which did not cause conflicts within the calls to the CLAPACK routines. However, when the process was parallelized within one system using threads conflicts arose due to the way threads run under the same memory space. Therefore, the mutexes where necessary in order to avoid the conflicts

causing erroneous results. Due to the fast creation of threads this expanded distribution within the cores of the system was expected to produce even greater reduction in the total computation time of the reconstruction. The actual results will be analyzed in the following chapter.

Chapter 4

Results and Conclusions

4.1 Introduction

The driving motivation of performing the research for the distributed reconstruction of the EKG was to improve the computation times. Therefore, in order to determine the benefits from the distribution, comparisons need to be made between the run times of the original and the redesigned reconstruction algorithms. Initially, reconstruction attempts conducted by the BPG on their hardware normally consisted of run times in the order of five minutes. However, to provide an accurate speed-up analysis all tests must be run on comparable hardware. As a result, the testing consisted of running all algorithms on the Virgo 2 cluster. In order to obtain an accurate representation of the execution times, the tests were performed when the system contained no other traffic. Tests consisted of repeated runs and obtaining execution times for each. Repeated runs were used to ensure the consistency of the results. The final time representation that is considered is the average time of all runs. It is also important to mention that reconstruction results were verified between the original algorithm and the redesigned implementations. Extra care was taken to ensure the implementations in C used the same mathematical approach that the original algorithm used. The results obtained from all C implementation were identical matches to those obtained from the original algorithm. These results and their meaning are explained in greater detail by Morales in [2]. As a result, the actual reconstruction results are not presented. Instead, the focus is kept on the benefits obtained from the distribution and the meaning of the results. This includes the speed-up for each version, and the maximum efficiency for the distributed approaches.

4.2 Original Algorithm

First, the original algorithm was run using the Matlab software. After running the original algorithm 10 times, the average run time in micro-seconds was 70,221,573. This is approximately 1 minute with 10 seconds, and it is a considerable speed-up from the original time of 5 minutes that it was taking the BPG. This speed-up is clearly attributed to the superior computational power possessed by Virgo 2 over standard personal computers that were being used by the BPG. Table 1 below shows the breakdown of times for each of the runs that was performed for the original algorithm. Further, it shows the consistency of the times and it demonstrates how the average time that will be used in comparison with the distributed versions is an accurate representation of the actual run time.

Table 1: Original Algorithm Run Times

Run	**1**	**2**	**3**	**4**	**5**	
Time (us)	70302809	70618355	70249822	69896155	70040726	
Run	**6**	**7**	**8**	**9**	**10**	**Average**
Time (us)	69856576	70069697	70215106	70489901	70101532	**70221573**

Despite the drastic decrease in time between the runs on Virgo 2 and those performed on the BPG hardware, there was still plenty of room for improvement through the use of distributed programming.

4.3 Serial Redesign

Before jumping to the results of the distributed approach, the serial version of the reconstruction algorithm implemented in C was tested. Similar to the tests performed for the original algorithm, the tests for the serial version consisted of 10 runs. The average run time for the serial version was 1,730,795 micro-seconds which is an improvement factor of 40.571. Table 2 below shows the individual run times for the tests performed on the serial version of the algorithm.

Table 2: Serial Redesign Run Times

Run	**1**	**2**	**3**	**4**	**5**	
Time (us)	1727687	1735167	1721620	1730211	1728342	
Run	**6**	**7**	**8**	**9**	**10**	**Average**
Time (us)	1736699	1731274	1732059	1726601	1738292	1730795

As expected, the serial implementation performed better than the original algorithm. The drastic improvement in performance can be attributed to addressing the inefficiencies present in the original algorithm which were described in the previous chapter. By simply addressing the inefficiencies the run times were scaled down from a degree of minutes to just a couple of seconds. This drop in run time can also be partially attributed to eliminating the overhead that is associated with an application like Matlab. Additional decreases in run time were still expected from the distributed approach.

4.4 Distribution Using Fork

After testing the original algorithm and the serial version in C, it was then time to test the distributed implementations. The first of the serial implementations tested was the one that used the *fork* system call to parallelize the remote procedure calls. These tests consisted of runs starting with only one compute node and progressively adding compute nodes until all 23 available nodes from the Virgo 2 cluster were used. Although, 2 of these nodes differed in that they were memory nodes, meaning that they contained 64 GB of memory, or 8 times as much memory as the compute nodes. However, preliminary tests showed that the additional available memory in the memory nodes bared no advantage in the computation times as it relates to the EKG reconstruction algorithm. The table below shows the average run times for each of the runs on the different number of nodes.

Table 3: Fork Distribution Run Times

Nodes	**1**	**2**	**3**	**4**	**5**	**6**
Run Time (us)	1553472	782266	524411	397117	319816	269080
Nodes	**7**	**8**	**9**	**10**	**11**	**12**
Run Time (us)	232777	205712	184653	168459	154706	143322
Nodes	**13**	**14**	**15**	**16**	**17**	**18**
Run Time (us)	134437	126458	119388	113541	108373	103729
Nodes	**19**	**20**	**21**	**22**	**23**	
Run Time (us)	99515	96303	93006	89303	87148	

Throughout the runs the parallel approach outperformed the serial implementation and thus the original algorithm. From the data it can be seen that initially the speed-up obtained from adding compute nodes was fairly linear, but as more nodes were added the speed-up began to

flatten out. However, increases in speed-up, or reductions in the run times, where accomplished through all additions of compute nodes. The fastest run time obtained from the distribution using *fork* was of 87148 micro-seconds. This reflects a decrease in run time by a factor of 19.86 over the serial redesign and an even bigger decrease of run time by a factor of 805.77 over the original algorithm. *Figure 5* below better illustrates the reduction in run times that were obtained.

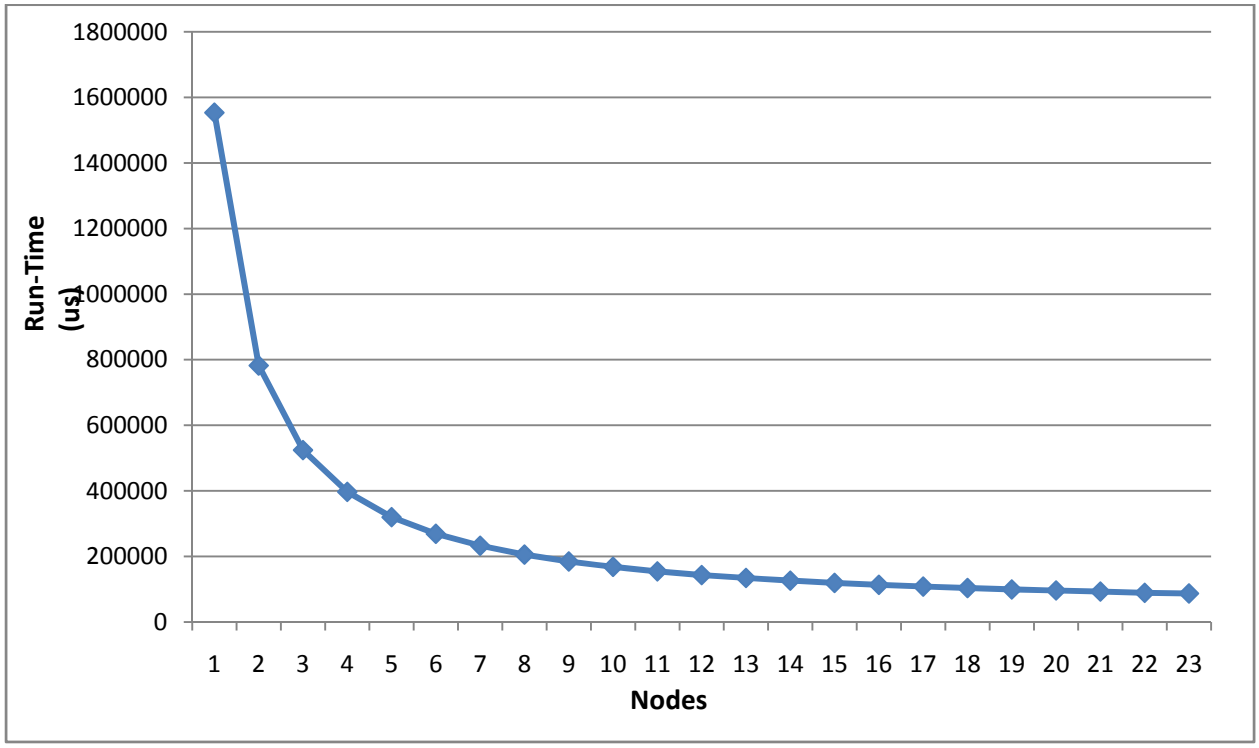

Figure 5: Fork Distribution Run Times

Next, in *Figure 6*, we can see the speed-up generated by the addition of each node into the distributed algorithm. As the graph shows, an increase in speed-up was obtained throughout the tests by continuing to add nodes, but a closer look at the actual speed-up shows that initially the speed-up obtained was more linear with respect to the number of nodes. For example, when 2 nodes where used the speed-up obtained was close to 2, but when all 23 nodes are used the speed-up barely approaches 18.

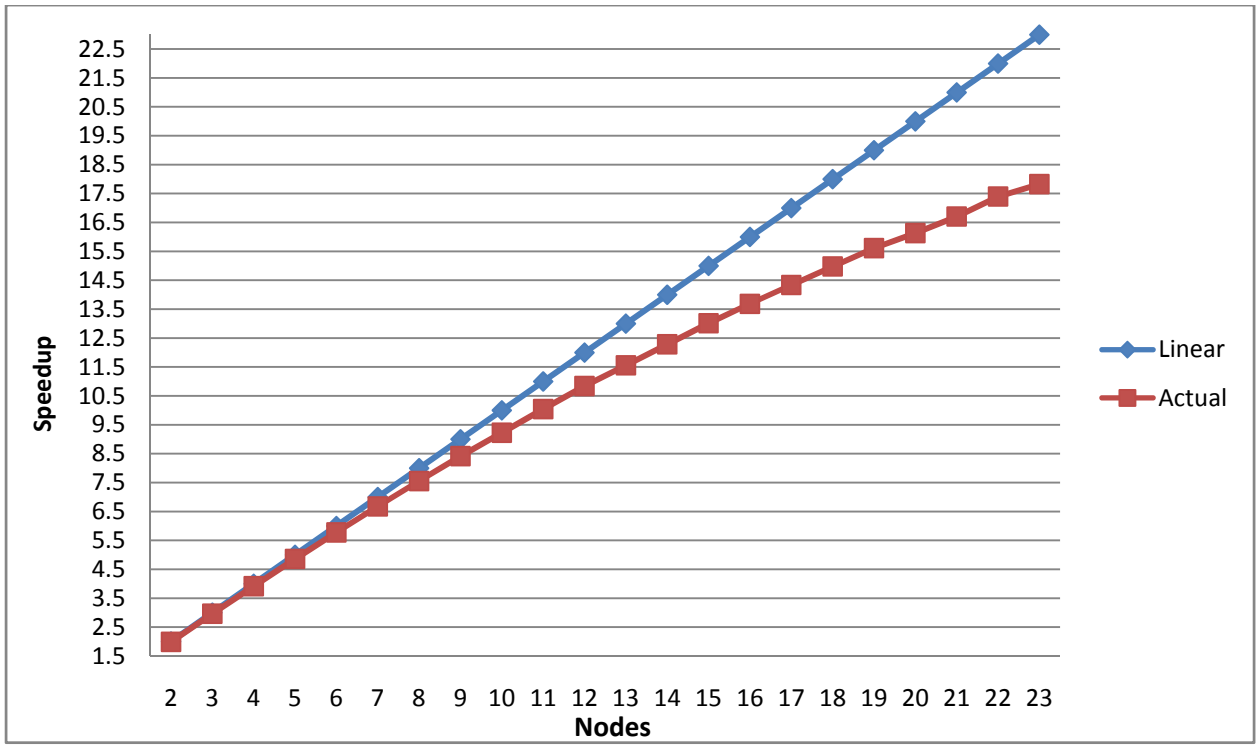

Figure 6: Fork Distribution Speed-up

Figure 6 also shows a graph illustrating the speed-up that would have been possible if a linear relationship existed between the speed-up and the number of nodes used. Comparing the two graphs also helps to illustrate how close to linear the speed-up obtained when fewer nodes were used as opposed to the flattening out seen when more nodes are used. The table below shows the values plotted in *Figure 6* above.

Table 4: Fork Distribution Speed-up

Nodes	**2**	**3**	**4**	**5**	**6**	**7**
Speed-up	1.99	2.96	3.91	4.86	5.77	6.67
Nodes	**8**	**9**	**10**	**11**	**12**	**13**
Speed-up	7.55	8.41	9.22	10.04	10.84	11.56
Nodes	**14**	**15**	**16**	**17**	**18**	**19**
Speed-up	12.28	13.01	13.68	14.33	14.98	15.61
Nodes	**20**	**21**	**22**	**23**		
Speed-up	16.13	16.7	17.4	17.83		

To better understand the implications of the speed-up, the efficiency is calculated for each speed-up value. The higher the efficiency translates to the best use of system resources. *Figure 7* shows a graph of the efficiency calculations to correspond to the speed-up values from above.

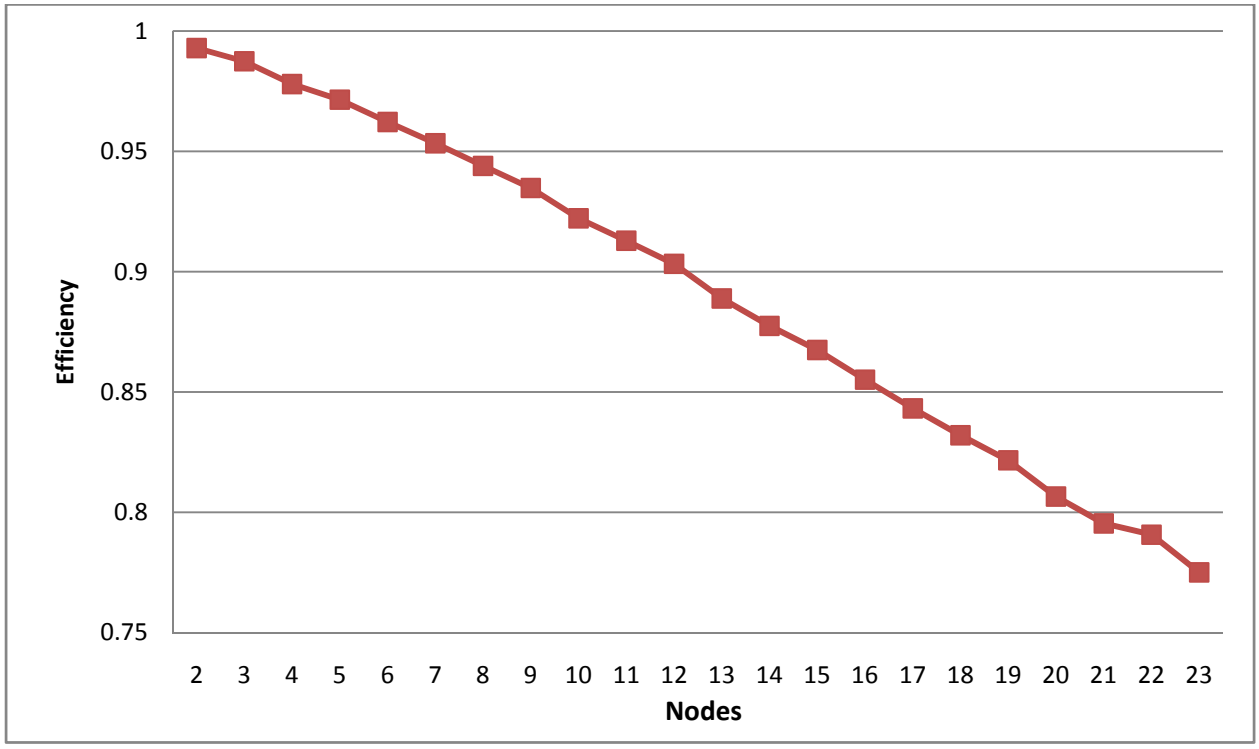

Figure 7: Fork Distribution Efficiency

These efficiency values show that overall the system resources are well utilized throughout all the runs with a smallest efficiency of 77.5% being produced when all 23 nodes were used. Surprisingly, implementing only 2 nodes generated the largest efficiency values of 99.2 %. However, this does not indicate that implementing only 2 nodes would be best suited for this type of distribution because it only provides a speed-up of 1.99. A trade-off needs to be made between the speed-up values and the efficiency that they provide. A better implementation would be represented by using 7 or 8 nodes where the speed-up is in the vicinity of 7 and the efficiency is still very respectable at approximately 95%. The table below shows the values plotted in *Figure 7*.

Table 5: Fork Distribution Efficiency

Nodes	**2**	**3**	**4**	**5**	**6**	**7**
Efficiency	0.992931	0.98744	0.977969	0.971478	0.962212	0.953378
Nodes	**8**	**9**	**10**	**11**	**12**	**13**
Efficiency	0.943959	0.934768	0.922167	0.912858	0.903256	0.8888788
Nodes	**14**	**15**	**16**	**17**	**18**	**19**
Efficiency	0.877467	0.867463	0.855127	0.843207	0.832017	0.8216016
Nodes	**20**	**21**	**22**	**23**		
Efficiency	0.806551	0.795379	0.790705	0.775026		

4.5 Distribution Using Threads

Using the *fork* system call to distribute the reconstruction greatly benefited in the area of reduced run times. However, due to the known advantages of using multi-threading, better run times were expected from the distribution using threads. Like the previous distributed tests, the test of threaded distribution consisted of runs beginning with only 1 node and working up to 23 nodes. The table below shows the times obtained from the distributed runs using threads.

Table 6: Thread Distribution Run Times

Nodes	**1**	**2**	**3**	**4**	**5**	**6**
Run Time (us)	1548865	781978	524261	395761	318512	267447
Nodes	**7**	**8**	**9**	**10**	**11**	**12**
Run Time (us)	231375	203705	181765	165409	151307	140550
Nodes	**13**	**14**	**15**	**16**	**17**	**18**
Run Time (us)	130331	121721	115244	108605	103353	98348
Nodes	**19**	**20**	**21**	**22**	**23**	
Run Time (us)	94175	90217	87058	83615	80882	

These figures show that the fastest computation time using threads was obtained when using 23 nodes for the distribution which produced a run time of only 80,882 micro-seconds. The thread distribution, as expected, outperformed the fork distribution by close to 7,000 microseconds. *Figure 8* below shows a graphical representation of the run time to provide a visualization of how run times compared among the different number of nodes.

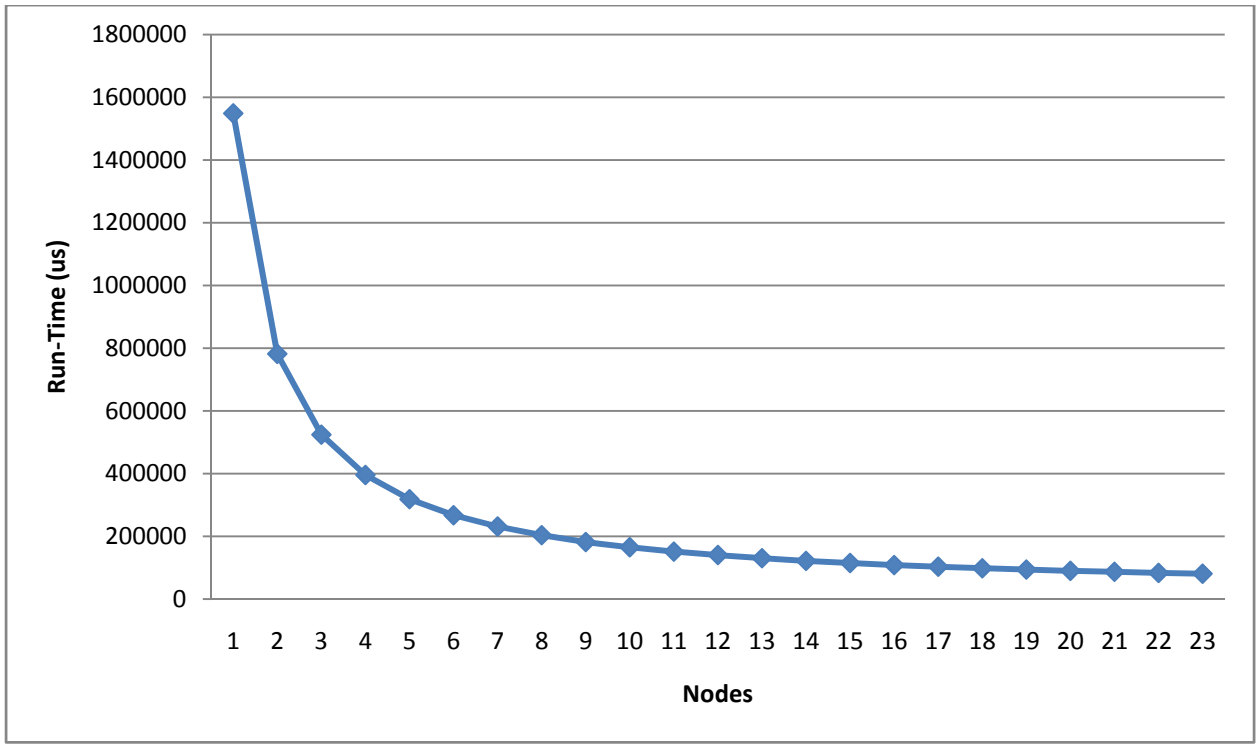

Figure 8: Thread Distribution Run Times

The figure shows that similar to the previous distributed implementation, the threaded distributed implementation generated reduced run times for each additional node that was added to the computations. *Figure 9* below shows the speed-up as it relates to the threaded distribution for the different number of nodes used. Although the speed-up increases continually as nodes are added, the relationship between them is not linear. During the use of a smaller number of nodes the relationship approaches linearity, but as the number of nodes increases the relationship diverts from linearity and the speed-up graph flattens out. The flattening out is representative of a loss of efficiency of the use of system resources.

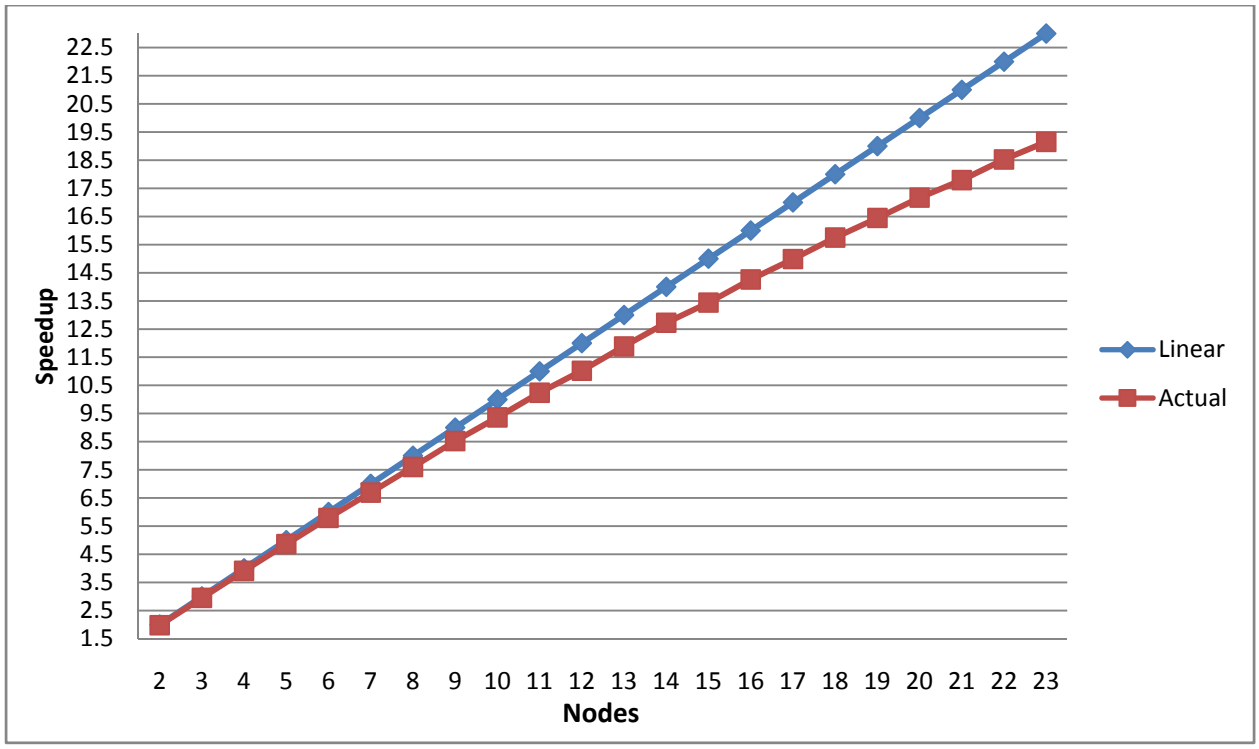

Figure 9: Thread Distribution Speed-up

The table below shows the actual values for the points plotted in the *Figure 9* above. From these values we can see that the maximum speed-up obtained from the threaded distribution is 19.15.

Table 7: Thread Distribution Speed-up

Nodes	**2**	**3**	**4**	**5**	**6**	**7**
Speed-up	1.980701	2.954376	3.913632	4.862821	5.791302	6.69418
Nodes	**8**	**9**	**10**	**11**	**12**	**13**
Speed-up	7.603479	8.521266	9.363869	10.23656	11.02002	11.88413
Nodes	**14**	**15**	**16**	**17**	**18**	**19**
Speed-up	12.72471	13.43987	14.26141	14.98617	15.74878	16.44668
Nodes	**20**	**21**	**22**	**23**		
Speed-up	17.16827	17.7911	18.5237	19.14959		

Finally, in order to obtain a better understanding of the speed-up associated with the use of each number of nodes, the efficiency was calculated for each. *Figure 10* below shows the efficiency graph corresponding to the number of nodes used in the threaded distribution.

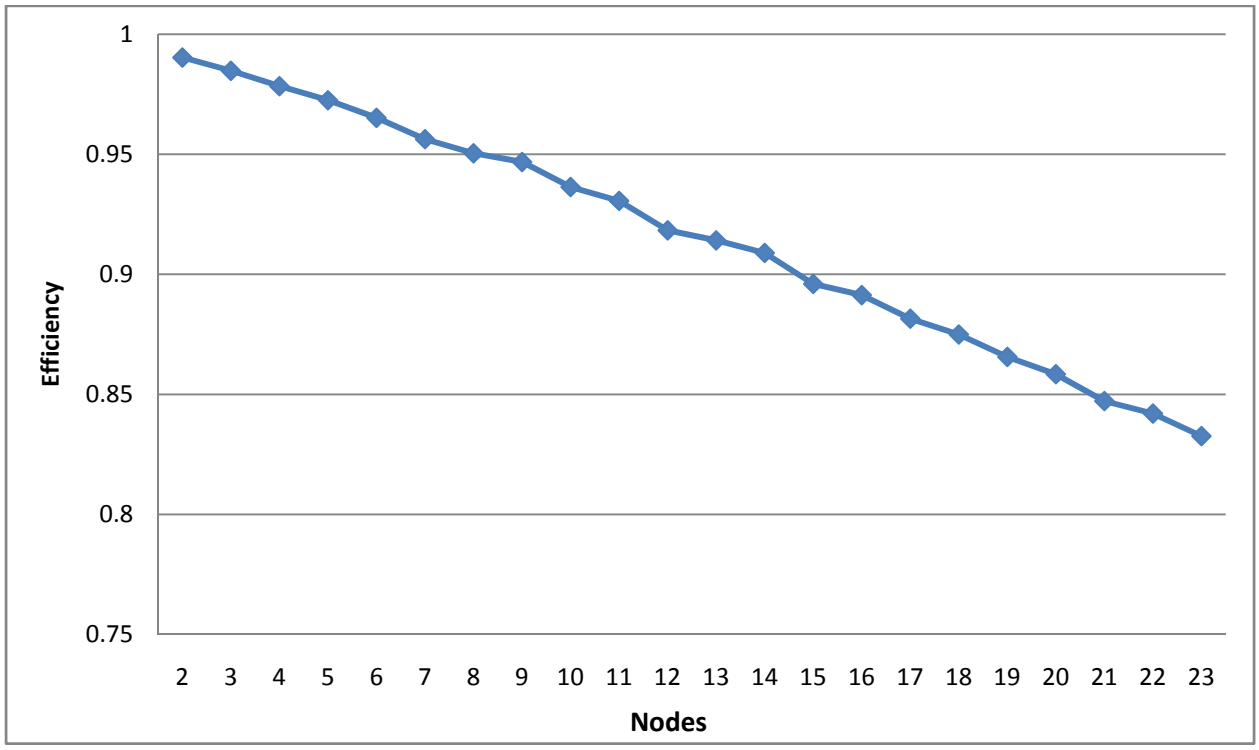

Figure 10: Thread Distribution Efficiency

This figure shows that the threaded distribution also generated good efficiency rating with the worst being when 23 nodes were used at a value of 83.25%. In this distribution the best efficiency was also generated by the use of only 2 nodes with a value of 99 %. Again, using a trade-off of speed-up vs. efficiency the best choice of node usage would be using 8 or 9 nodes that would provide a speed-up of 8 and a run time of about 190,000 us. The efficiency values plotted in *Figure 10* are shown in the table below.

Table 8: Thread Distribution Efficiency

Nodes	**2**	**3**	**4**	**5**	**6**	**7**
Efficiency	0.99035	0.984792	0.978408	0.972564	0.965217	0.956311
Nodes	**8**	**9**	**10**	**11**	**12**	**13**
Efficiency	0.950435	0.946807	0.936387	0.930596	0.918335	0.914164
Nodes	**14**	**15**	**16**	**17**	**18**	**19**
Efficiency	0.908908	0.895991	0.891338	0.88154	0.874932	0.865615
Nodes	**20**	**21**	**22**	**23**		
Efficiency	0.858413	0.847195	0.841986	0.832591		

4.6 Server Threads

The thread distribution described in the previous section consisted of implementing threads on the client code to parallelize the remote calls. This section covers the tests performed when threads were implemented on both the client and server. The idea for using threads in the server also grew from the fact that each server node consists of 8 cores. Therefore, it was expected that further breaking down the reconstruction using threads would lead to benefits such as lower run times and higher efficiency. The table below shows the run time values obtained from the tests performed on a increasing number of nodes.

Table 9: Server Side Thread Run Times

Nodes	**1**	**2**	**3**	**4**	**5**	**6**
Run Time(us)	1620638	816733	556247	418772	338757	281308
Nodes	**7**	**8**	**9**	**10**	**11**	**12**
Run Time(us)	245168	216244	198661	174152	160611	150067
Nodes	**13**	**14**	**15**	**16**	**17**	**18**
Run Time(us)	137490	129763	123677	117518	112460	105837
Nodes	**19**	**20**	**21**	**22**	**23**	
Run Time(us)	103917	95599	91908	90290	84741	

Contrary to expectations the server side thread implementation did not produce faster run times. The fastest execution time of 84,741 micro-seconds resulted from using 23 nodes which is greater than the value obtained from the original thread distribution of 80,882 micro-seconds. The original thread distribution was consistently faster than the server thread implementation. The speed-up and efficiency study of this version of the distribution was comparable to that generated with the original thread distribution. Due to the unexpected results from this server thread implementation additional timing studies were conducted to attempt to identify the source

of delay. Two key factors were identified that caused the larger than expected run times. First, when implementing the QR factorization memory is dynamically allocated to use as buffers to store the large matrices that are being worked with. The server thread implementation caused this memory allocation to occur 8 times (once per thread). When threads were not used on the server the memory allocation only occurred once. The tests conducted showed that on average the time taken to allocate the memory was 79.2 micro-seconds. Since each of the 8 threads allocated memory there was an average of 633.6 micro-seconds per server added to the run time. The studies also showed that the timing for allocating memory was not very consistent. At times the memory allocation only took 15 micro-seconds, and at other times the time spiked to 500 micro-seconds. The second issue that contributed to the additional time associated with server side threads was also related to performing the QR factorization required in the reconstruction. Tests showed that only one thread could access any of the CAPLACK functions at a time, otherwise memory interference would occur. To deal with the issue a mutex was used to protect the area of code that performed the QR factorization. As a result, true parallelism was not possible because threads needed to perform the QR factorization one at time.

4.7 Conclusion

As intended the work presented in this thesis was able to provide the BPG with drastically reduced run times through several factors. First, it demonstrated the benefits and advantages of using the available resources such as the Virgo 2 cluster. Simply by using the cluster to perform the original algorithm, the run time was cut by more than half from runs that exceeded 5 minutes to an average run time of 1 minute 10 seconds. Improvements to the reconstruction algorithm were also generated by exposing and addressing computational inefficiencies in the original algorithm. Addressing these inefficiencies further improved the run times by a factor of 40. The last factor involved in providing reduced run times was related to the distributed programming implemented using RPC's. Incorporating distributed programming was able to reduce the run times in comparison to the original algorithm by an astonishing factor of 868. That is, a task that was originally taking over a minute to run completed in a fraction of a second (less than one-tenth of a second).

Most of the results obtained from the research done in this thesis were expected and fairly easy to hypothesize. For example, it was expected and confirmed that the redesigned C code would perform better than the original algorithm, and that the distributed versions would yield even better results. However, certain results obtained were contrary to what was expected. First, it was expected that efficiency would begin low when few nodes are used and increase as more nodes were added to the computations. However, the results showed that the efficiency as it relates to the problem of distributing the reconstruction of an EKG begins very high (at 99%) and tapers off as more nodes are added. Additionally, the level of efficiency obtained with every number of nodes implemented was surprising given that the lowest efficiency obtained was of 83%. Previous studies relating to the distribution of different problems have shown efficiency to

begin at about 50% and reach a maximum of about 80% [9]. The high level of efficiency obtained is attributed to the amount of parallelism present in the algorithm used in the reconstruction of an EKG.

Another unexpected result was found in the code that incorporated threads on both the client and server sides. It was expected that using threads on the server side would take further advantage of the available processing resources, and provide the lowest run times, but this was not the case. Tests showed that threads on both the client and server performed consistently slower than multi-threading only the client. These unexpected results were attributed to added overhead due to additional memory allocation required through the use of multi-threading. Additionally, the functions implemented in the reconstruction prevented true parallelization because mutexes were required to protect them. These results show that not every application can benefit from multi-threading.

4.8 Future Work

Future work can include implementations that provide a better interface for use by people who are not familiar with command line programs. Also, this work covered only reconstruction associated with a fixed dipole model of the heart. In parallel with future work which is anticipated from the BPG, research can be done for implementing distributed reconstructions that are based on a moving dipole model of the heart. Additionally, interest has been expressed in developing reconstruction algorithms that incorporate multiple dipoles. Research in the reconstruction based on moving and multiple dipoles can lead to slightly different results from those obtained through this research. In addition, other methods of implementing the distribution aside from RPC's can be experimented with. The work presented in this thesis provides a foundation in the distributed reconstruction of EKG signals, but additional work can continue to aid the cause of reconstruction.

References

1. Tanenbaum, Andrew S. Modern Operating Systems. Second Edition. New Jersey. Pretince-Hall. 2001.
2. Morales, Eduardo, Pierluissi, Joseph. Implementation of the Moving Dipole Model of the Human heart to the Reconstruction of Measured Precordial Leads, The University of Texas at El Paso, United Sates – Texas. Medical and Engineering Publishers. 2008.
3. P.W. Macfarlane, L. Edenbrandt, O. Pahlm. 12-Lead VectorCardiography. Butterworth-Heinemann. 1995.
4. Hairong Kuang, Lubomir F. Bic, Michael B. Dillencourt and Adam C. Chang, "PODC: Paradigm-Oriented Distributed Computing," Distributed Computing Systems, 1999. Proceedings. 7th IEEE Workshop Future Trends of. Dec 20-22, 1999, pp 169-175.
5. Bonnie H. Bennett, Emmet Davis and Timothy Kunau, "Beowulf Parallel Processing for Dynamic Load-balancing," Aerospace Conference Proceedings, 2000 IEEE. Volume 4, March 18-25, 2000, pp 389-395.
6. P.H. Carns, W.B Ligon III, S.P. McMillan and R.B. Ross, "An Evaluation of Message Passing Implementations on Beowulf Workstations," Aerospace Conference, 1999. Proceedings. 1999 IEEE. Volume 5, March 6-13, 1999, pp 41.45
7. Stevens, Richard W., UNIX Network Programming: Interprocess Communications, Volume 2, Second Edition, Upper Saddle River, NJ: Prentice Hall PTR, 1999.
8. Bach, Maurice J.The Design of the UNIX Operating System, New Jersey, Prentice Hall, 1990.

9. Cordova, Hector. "Analysis of a Parallelized Neural Network Training Program Implemented Using MPI and RPCs." Master's Thesis. The University of Texas at El Paso. El Paso, TX. 2008.
10. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D. LAPACK Users' Guide. Third Edition. Philadelphia, PA. Society for Industrial and Applied Mathematics. 1999.
11. Abd-El-Barr, Mostaffa, El-Rewini, Hesham. Fundamentals of Computer Organization and Architecture. New Jersey. John Wiley & Sons. 2005.
12. "The MathWorks- Accelerating the pace of engineering and science". Matlab. 25 Nov. 2008 <http://www.mathworks.com/access/helpdesk/help/techdoc/index.html?/access/helpdesk/help/techdoc/ref/mrdivide.html >

Appendix A

Source Code

A.1 Conversion Code (*convert.c*)

The following is the source code of the program that converted the data from the Matlab format to a file that could be read by a C program. This program name *convert.c* is designed to be compiled and used as function in Matlab which explains the inclusion of the header file *mex.h*, and the lack of a *main* function which is replaced by a *mexFunction*.

```
/*
 * This program will create a raw data file to be read by a C program
 *  from a data element provided as an argument. File name is also passed
 *  as a argument.
 *
 *  First argument can be a scalar or a matrix with up to 3 dimensions.
 */

#include <stdio.h>
#include <fcntl.h>
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]){
   mxArray *data;
   int *Dims;                  /*pointer to an array to hold dimensions of matrix*/
   int nDimensions;            /*Number of dimensions in matrix*/
   int sizeString;             /*size of string which is the filename*/
   int i,j,k;                  /*Generic counters*/
   int fd;                     /*file descriptor*/
   int err;                    /*used in error handling*/
   char *filename;             /*pointer to array to hold filename*/
   double *matrix;             /*pointer to matrix passed*/
   double dummy;               /*dummy double dummy used in buffer for writing*/

   if(nrhs != 2){              /*verify two arguments passed*/
      mxErrMsgTxt("This Function takes two arguments\nUsage:  convert(arg1,'filename')\n");
   }
   if(nlhs > 0){               /*verify call to function does not expect a return*/
      mxErrMsgTxt("This Function does not return values\nUsage:   convert(arg1, 'filename')\n");
   }
   /*verify second argument is a string*/
   if(mxGetM(prhs[1]) != 1 || !mxIsChar(prhs[1])){
      mxErrMsgTxt("Second Argument must be a string");
   }
   sizeString = mxGetN(prhs[1]) + 1;   /*find number of chars in string +1 for null terminator*/
   /*allocate memory for string*/
   filename = mxCalloc(sizeString, sizeof(char));
   /*obtain second argument with error checking*/
   if(mxGetString(prhs[1], filename, sizeString) == 1){
     mxErrMsgTxt("Error Reading File Name");
   }
   printf("File name has been extracted as %s\n", filename);
```

```
   data=prhs[0];
   nDimensions = mxGetNumberOfDimensions(data);  /*find # of dimensions of matrix*/
   printf("The passsed argument has %d Dimensions\n", nDimensions);
   Dims = malloc(nDimensions * sizeof(int)); /*allocate memory for # of dimensions*/
   for(i=0; i< nDimensions; i++){
     Dims[i]  = mxGetDimensions(data)[i];
     printf("Dimension %d = %d\n", i+1, Dims[i]);
   }

   if((fd=open(filename, O_CREAT | O_WRONLY, 0777))<=0){
      mxErrMsgTxt("Error Creating File %s ", filename);
   }

   matrix = mxCalloc(mxGetElementSize(data), mxGetNumberOfElements(data));
   matrix = mxGetData(data);

   if(nDimensions < 3){
      for(i=0;i<Dims[0];i++){
         for(j=0;j<Dims[1];j++){
/*             dummy = matrix[j*Dims[0]+i];          used to store in row-major form*/
            dummy = matrix[i*Dims[1]+j];          /*used to store in col major form*/
            err = write(fd, &dummy, sizeof(double));
            if(err != sizeof(double)){
               printf("error writing byte i = %d, j = %d", i, j);
               return;
            }
         }
      }
   } else {
      for(k=0;k<Dims[2];k++){
         for(i=0;i<Dims[0];i++){
            for(j=0;j<Dims[1];j++){
/*                dummy = matrix[(k*Dims[0]*Dims[1])+(j*Dims[0])+i]; used to store in row-major form*/
               dummy = matrix[(k*Dims[0]*Dims[1])+(i*Dims[1])+j];
               err = write(fd, &dummy, sizeof(double));
               if(err != sizeof(double)){
                  printf("error writing byte k = %d, i = %d, j = %d", k, i, j);
                  return;
               }
            }
         }
      }
   }
   close(fd);
   return;
}
```

A.2 Serial Reconstruction Code (*Reconstruct.c*)

The following source code, *Reconstruct.c*, was used to implement the EKG reconstruction in C. It is a serial implementation used to determine what benefits were obtained from addressing the inefficiencies present in the original algorithm.

```
/*****************
*
*  This program is the serial implementation of the
*   EKG reconstruction algorithm
**********************/
#include <stdio.h>
#include "f2c.h"
#include "blaswrap.h"
#include <fcntl.h>
#include <math.h>
#include <sys/time.h>

main(int argc, char **argv){
        struct timeval starttime,                       //struct to hold time variables
                       endtime;
        int fd;                                                 //hold file descriptor
        int err;                                        //holds error codes for system calls
        int i,j,k,l, count;                             //general counter
        int startIND, endIND;
        int index;
        double hold1, hold2;
        double matrix[3][20571][14];                    //buffer for lead vector matrix
        double patient[12][302];                        //buffer for patient matrix
        double *mat, *pat;                              //pointer matrices for data from files
        double *A, *B, *TAU, *WORK, *WORKB;                     //pointer matrixes
        double *M, *L, *C, *NEW;                        //pointer to matrices for dgemm
        double ALPHA = 1.0, BETA=0.0;                           //scalars for dgemm
        double *pivot;                          //pivoting matrix
        integer ROWA=6, COLA=3;                                 //scalar arguments for dgeqp3
        integer INFO, LDA=ROWA;                                 //scalar arguments
        integer ROWB=6, COLB=302;                               //scalar arguments for dormqr
        integer LDB=ROWB;                                       //leading vector length for matrix B
        integer LWORK = 3 * COLA+1;                             //work array length for dgeqp3
        integer LWORKB = COLB;                                  //Work array length for dormqr
        integer JPVT[3];                                //pivoting array for dgeqp3
        integer ROWM=6, COLN=302, ROWK=3;                       //dimensions for dgemm
        integer LDC=ROWM;
        char SIDE = 'L', TRANS = 'T';                           //character arguments for
        char UPLO = 'U', TRANST = 'N', DIAG = 'N';      //character arguments for dtrtrs function
        char TRANSA = 'N', TRANSB = 'N';                //character arguments for dgemm
        char str[10];                                           //temp string
/* The following variables are strictly used for error calculation */
      double MaxAbs[6];
      double NormXact[6][302];
      double NormApp[6][302];
        double Error[6][302];
        double RMSerr[6];
        double SumErr, MinErr = 9999;
        int MinInd;
```

```
mat = (double *) malloc(sizeof(matrix));		//Allocate space for pointers
pat = (double *) malloc(sizeof(patient));
A = (double *) malloc(6*3*sizeof(double));
C = (double *) malloc(6*3*sizeof(double));
B = (double *) malloc(6*302*sizeof(double));
M = (double *) malloc(3*302*sizeof(double));
L = (double *) malloc(3*302*sizeof(double));
TAU = (double *) malloc(3*sizeof(double));
WORK = (double *) malloc(LWORK*sizeof(double));
WORKB = (double *) malloc(LWORKB*sizeof(double));
NEW = (double *) malloc(ROWM*COLN*sizeof(double));
pivot = (double *) malloc(3*3*sizeof(double));

if(argc != 3){					//Verify starting and ending index are included as parameters.
	printf("Must provide starting and ending Index\n");
	return 0;
}
startIND = atoi(argv[1]);			//use ascii to integer function to save starting and ending indexes
endIND = atoi(argv[2]);


gettimeofday(&starttime, 0);			//start timing
//open file and check for errors
fd = open("data/LeadVectorC.d", O_RDONLY);
if(fd < 0){
	printf("Error %i opening file data/LeadVectorC.d", fd);
	return 0;
}
//read file and check for errors
err = read(fd, &matrix, sizeof(matrix));
if(err <= 0){
	printf("Error %i reading file data/LeadVectorC.d", err);
	return;
}
close(fd); //close file
//open Patient file and check for errors
fd = open("data/NormalPatientC.d", O_RDONLY);
if(fd <= 0){
	printf("Error %d opening data/NormalPatientC.d\n", fd);
	return;
}
//Read Patient File
err = read(fd, &patient, sizeof(patient));
if(err<=0){
	printf("Error %d reading data/NormalPatientC.d\n", fd);
	return;
}
close(fd);

//set pointers for lead vector matrix and original patient matrix
mat = (double *) matrix;
pat = (double *) patient;
//initilaize MaxAbs array to be used in condition statement when calculation Normalization
for(i=0;i<6;i++){
	MaxAbs[i] = 0;
}

//enter main loop that interates from startIND to endIND (arguments obtained from command line)
for(index=startIND;index<endIND;index++){
	for(l=0;l<3;l++){
		JPVT[l]= 0;
```

```
}
//copy patient matrix values to new memory region to conserve values
for(i=0;i<302;i++){
        for(j=0;j<6;j++){
                B[i*6+j] = pat[i*12+j+6];
        }
}
count = 0;
for(j=0;j<3;j++){
        for(k=0;k<6;k++){
        A[count] = mat[(j*287994)+(k*20571)+index];    //store lead vector values in A to preserves
        C[count] = mat[(j*287994)+(k*20571)+index];    //store matrix in C for later use
        count++;
        }
}

/* call clapack routine to obtain QR factorization of A. A is overwritten with orthogonal matrix Q and
triangular matrix R*/
dgeqp3_(&ROWA, &COLA, A, &LDA, JPVT, TAU, WORK, &LWORK, &INFO);

//generate pivot matrix later used to obtain dipole moment as part of the equation A*P = Q*R where P is the
pivot matrix
for(i=0;i<3;i++){
        for(j=0;j<3;j++){
                if(j==JPVT[i]-1){
                        pivot[i*3+j] = 1.0;
                } else {
                        pivot[i*3+j] = 0.0;
                }
        }
}

//use clapack routine to multiple orthogonal matrix Q' contained in A by lead vector matrix B
//B is overwritten with product
dormqr_(&SIDE, &TRANS, &ROWB, &COLB, &COLA, A, &LDA, TAU, B, &LDB, WORKB,
 &LWORKB, &INFO);

//the multiplication of the matrix A' which is 3x6 and the matrix B which is 6x302 results in a matrix
// of dimensions 3x302 but is stored in B which is 6x302 therefore it is migrated to M which is 3x302
for(j=0;j<302;j++){
        for(k=0;k<3;k++){
                        M[j*3+k] = B[j*6+k];
        }
}

//use clapack to solve R * M = (Q' * B) for M (the dipole moment) where R is the upper triangular matrix
generated by dgeqp3
//and the value Q'*B is the result of dormqr which is now contained in M
//the result overwrites M
dtrtrs_(&UPLO, &TRANST, &DIAG, &COLA, &COLB, A, &LDA, M, &COLA, &INFO);

//multiple the result M by the previously generated pivot matrix to align columns with actual solution
//the result which is the dipole moment is contained in matrix L
dgemm_(&TRANSA, &TRANSB, &COLA, &COLB, &COLA, &ALPHA, pivot, &COLA, M, &COLA,
&BETA, L, &COLA);

//attempt to reconstruct original signal using by multiplying the dipole moment L with the lead vector C
//resultant reconstructed EKG singal is stored into NEW.
dgemm_(&TRANSA, &TRANSB, &ROWA, &COLB, &COLA, &ALPHA, C, &ROWA, L, &COLA,
&BETA, NEW, &LDC);

//error calculation for the current approximation NEW matrix.
```

```
		//find Maximum Absolute Value of each row of the Normal Patient
		if(MaxAbs[0] == 0){ //ensures that for each reconstruction the normalization only occurs once.
			for(i=6;i<12;i++){  //indexing only the last 6 rows to avoid extra computations not needed
				hold2=0;
				for(j=0;j<302;j++){
					if(pat[j*12+i] < 0){
						hold1 = pat[j*12+i] * -1;
					}
					else
						hold1 = pat[j*12+i];
					hold1 = fabs(hold1);
					if(hold1 > hold2){
						hold2 = hold1;
					}

				}
				MaxAbs[i-6] = hold2;		//store maximum absolute values into array
			}
			//Normalize normal patient vectors for Normal Patient
			for(i=6;i<12;i++){
				for(j=0;j<302;j++){
					NormXact[i-6][j] = pat[j*12+i] / MaxAbs[i-6];
				}
			}
		}

		//Normalize current reconstructed vector using saved results from original patient normalization
		//also compute RMS error for each of the 6 vectors
		//sum all six RMS errors into SumErr which represents the total error for the reconstruction
		SumErr = 0;
		for(i=0;i<6;i++){
			RMSerr[i] = 0;
			for(j=0;j<302;j++){
				hold1 = NormXact[i][j] - (NEW[j*6+i] / MaxAbs[i]);
				hold1 = hold1 * hold1;
				RMSerr[i] = RMSerr[i] + hold1;
			}
			RMSerr[i] = sqrt(RMSerr[i]/302);
			SumErr += RMSerr[i];
		}

		//determine if error for current reconstruction is less that previous errors.
		//if so save error and index as possible solution
		if(SumErr < MinErr){
			MinErr = SumErr;
			MinInd = index;
		}



	}	//end main loop

	gettimeofday(&endtime, 0);		//obtain end time
	printf("Min Error = %9.20f at Index = %d\n", MinErr, MinInd);	//display results
	//display timing
	printf("\nRun Time = %1.0f us\n", ((endtime.tv_sec - starttime.tv_sec)*1000000.0 + (endtime.tv_usec -
	starttime.tv_usec)));
	return 0;

}
```

A.3 Parallel Reconstruction Code

The code used in the distributed implementations is presented here.

A.3.1 RPC Specification File (*Reconstruct.x)*

Below is the code that was used as the starting point for the distribution using RPC's. This specification file was used as the input to the *rpcgen* function which created the header file and client and server stubs that were used in all version of the distributed algorithm.

```
struct rec_in{          //holds input arguments
        int start_index;
        int end_index;
};

struct rec_out{         //holds output values
        double error;
        int index;
};

program RECONSTRUCT {
        version REC_VERS{
                rec_out qrerror(rec_in) = 1;
        }  = 1;
} = 0x31230000;
```

A.3.2 Fork Client File (*client.SHM.c)*

The client side source code that uses the *fork* system call to implement distribution is shown below.

```
#include <stdio.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/shm.h>
#include "Reconstruct.h"

main(int argc, char **argv){

        int i, shmid1, increment;
        rec_in input;
        rec_out *output;
        double *error;
        int *index;
        struct timeval starttime,
                        endtime;

        struct shared_area {//shared memory data layout
                pthread_mutex_t mutex;
                double error;
                int index;
```

```
} *locker;
typedef shared_area;
CLIENT *cl;
pid_t child[25];


gettimeofday(&starttime, 0);  //start time
//Create shared memory region
if((shmid1 = shmget(ftok("shpath", getpid()), sizeof(locker), 0600  | IPC_CREAT)) < 0){
        printf("Error creating shared memory region\n");
        return;
}
//Attach shared memory region
if((locker = shmat(shmid1, 0, 0)) < 0){
        printf("Error attaching shared memory region\n");
        return;
}
//initialize shared memory
locker->error = 999999;
locker->index = 0;

//calcualte increment values
increment = 20571 / (argc - 1);
if(20571%(argc-1) != 0)
        increment++;

//for loop to fork processes
for(i=0; i<argc-1; i++){
        if((child[i] = fork()) ==0){
                //set up start & end index for current proc
                if(i == 0)
                        input.start_index = 0;
                else
                        input.start_index = increment * i;
                if(i == argc - 2)
                        input.end_index = 20571;
                else
                        input.end_index = input.start_index + increment;
                //create client
                cl = clnt_create(argv[i+1], RECONSTRUCT, REC_VERS, "tcp");
                //call remote procedure
                output = qrerror_1(&input, cl);
                //set mutex and update error values if applicable
                pthread_mutex_lock(&locker->mutex);
                if(locker->error > output->error){
                        locker->error = output->error;
                        locker->index = output->index;
                }
                pthread_mutex_unlock(&locker->mutex);
                return 0;
        }
}
//wait for child processes
for(i=0; i<argc - 1; i++){
        waitpid(child[i], NULL, 0);
}
gettimeofday(&endtime, 0);   //end timing
//display results
printf("\nPARENT MIN ERROR = %9.20f\t\tINDEX = %d\n", locker->error, locker->index);
//detach and remove shared memory
if((shmdt(locker)) != 0){
        printf("Error detaching shared memory region\n");
```

```
		return;
	}
	if((shmctl(shmid1, IPC_RMID, NULL)) != 0){
		printf("Error removing shared memory region\n");
		return;
	}
	//display run times
	printf("\nRun Time = %1.0f us\n\n", ((endtime.tv_sec - starttime.tv_sec)*1000000.0 + (endtime.tv_usec -
	starttime.tv_usec)));
	return 0;
}
```

A.3.3 Thread Client File (*client.PTH.c)*

The code used for the client when threaded distribution was implemented is shown below.

```
#include <stdio.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/shm.h>
#include "Reconstruct.h"
#include <pthread.h>


struct thread_area{  //individual thread data
	CLIENT *cl;
	char * host;
	rec_in input;
	rec_out *output;
} threadArray[25];
typedef thread_area;
double MinErr = 99999;
int MinInd;
pthread_mutex_t mutex;

main(int argc, char **argv){

	int i, increment, err, j;
	struct timeval starttime,
			endtime;

	pthread_t thread[25];

	void thread_RPC(void *);	//prototype for thread function

	gettimeofday(&starttime, 0);  //get start time

	//calculate increment for indexes
	increment = 20571 / (argc - 1);
	if(20571%(argc-1) != 0)
		increment++;

	//loop to create threads
	for(i=0; i<argc-1; i++){
		//calculate starting and ending index for current thread
		if(i == 0)
			threadArray[i].input.start_index = 0;
		else
			threadArray[i].input.start_index = increment * i;
```

```
                if(i == argc - 2)
                        threadArray[i].input.end_index = 20571;
                else
                        threadArray[i].input.end_index = threadArray[i].input.start_index + increment;
                threadArray[i].host = argv[i+1];
                //create thread and call function
                if((err = pthread_create(&thread[i], NULL, (void *) thread_RPC, &threadArray[i])) != 0){
                        printf("Error creating thread #%d\n", i+1);
                        return 0;
                }
        }
        //wait for threads
        for(j=0; j<argc - 1; j++){
                err=pthread_join(thread[j], NULL);
        }
        gettimeofday(&endtime, 0);   //get end time

        //print results
        printf("\nPARENT MIN ERROR = %9.20f\t\tINDEX = %d\n", MinErr, MinInd);
        //print run times
        printf("\nRun Time = %1.0f us\n\n", ((endtime.tv_sec - starttime.tv_sec)*1000000.0 + (endtime.tv_usec -
        starttime.tv_usec)));
        return 0;
}

void thread_RPC(void * threadthread_args){
        struct thread_area *thread_args;
        thread_args = (struct thread_area *) threadthread_args;

        //create client
        thread_args->cl = clnt_create(thread_args->host, RECONSTRUCT, REC_VERS, "tcp");
        //call remote procedure
        thread_args->output = qrerror_1(&thread_args->input, thread_args->cl);
        //update Min and Max error
        if(thread_args->output->error < MinErr){
                MinErr = thread_args->output->error;
                MinInd = thread_args->output->index;
        }
        pthread_exit(NULL);
        return;
}
```

A.3.4 Serial Server File (*server.c)*

The code used on the server side for both the fork and thread distribution is shown below.

```
#include "Reconstruct.h"
#include <string.h>
#include "f2c.h"
#include "blaswrap.h"
#include <fcntl.h>
#include <math.h>

double matrix[3][20571][14];                    //buffer for lead vector matrix
double patient[12][302];                        //buffer for patient matrix
int bool_open = 0;
double *mat, *pat;

rec_out * qrerror_1_svc(rec_in *argsptr, struct svc_req *rqstp){
        static rec_out result;
```

```
int count, index, i, j, k;
integer CONST3 = 3,
        CONST6 = 6,
        CONST302 = 302;
char CONSTN = 'N',
    CONSTL = 'L',
    CONSTT = 'T',
    CONSTU = 'U';
integer INFO, JPVT[3];
double ALPHA = 1.0, BETA = 0.0;
double *B_hold;
double *A, *B, *C, *L, *M, *NEW, *PIVOT, *TAU, *WORK, *WORKB;
double hold1, hold2, NormApp[6][302], Error[6][302], RMSerr[6], SumErr, MinErr = 9999;
double MaxAbs[6];
double NormXact[6][302];
int MinInd;

void open_files();                      //prototype

mat = (double *) malloc(sizeof(matrix));
pat = (double *) malloc(sizeof(patient));
B_hold = (double *) malloc(6*302*sizeof(double));
A = (double *) malloc(6*3*sizeof(double));
B = (double *) malloc(6*302*sizeof(double));
C = (double *) malloc(6*3*sizeof(double));
L = (double *) malloc(3*302*sizeof(double));
M = (double *) malloc(3*302*sizeof(double));
NEW = (double *) malloc(CONST6*CONST302*sizeof(double));
PIVOT = (double *) malloc(3*3*sizeof(double));
TAU = (double *) malloc(3*sizeof(double));
WORK = (double *) malloc(CONST302*sizeof(double));
WORKB = (double *) malloc(CONST302*sizeof(double));

open_files();
if(bool_open == 0){             //determine if files have been opened. if not call function to open files.
        bool_open = 0;
}

//use pointers to access lead vector and original patient data.
mat = (double *) matrix;
pat = (double *) patient;
//copy patient data into HOLD matrix to be used later to refresh patient data
for(i=0;i<302;i++)
        for(j=0;j<6;j++)
                B_hold[i*6+j] = pat[i*12+j+6];
//initialize MaxAbs to be used later in condition statement when creating normalized patient data
MaxAbs[0] = 0;
MinErr = 99;

for(index=argsptr->start_index;index<=argsptr->end_index;index++){
        //initialize JPVT pointer
        for(i=0;i<3;i++)
                JPVT[i] = 0;
        memcpy(B, B_hold, 6*302*sizeof(double));//refresh original patient matrix B from hold matrix
        count = 0;
        for(i=0;i<3;i++){
                for(j=0;j<6;j++){
                        A[count] = mat[(i*287994)+(j*20571)+index];    //store lead vector data into
                                //matrix A to preserve integrity of data
                        C[count] = mat[(i*287994)+(j*20571)+index];    //store matrix in C for later
                                //use because A will be overwritten
                        count++;
```

```
		}
}
/* call clapack routine to obtain QR factorization of A. A is overwritten with orthogonal matrix Q
and triangular matrix R*/
dgeqp3_(&CONST6, &CONST3, A, &CONST6, JPVT, TAU, WORK, &CONST302, &INFO);

//generate pivot matrix later used to obtain dipole moment as part of the equation A*P = Q*R
where P is the pivot matrix
for(i=0;i<3;i++)
		for(j=0;j<3;j++)
			if(j==JPVT[i]-1)
				PIVOT[i*3+j] = 1.0;
			 else
				PIVOT[i*3+j] = 0.0;

//use clapack routine to multiple orthogonal matrix Q' contained in A by lead vector matrix B
//B is overwritten with product
dormqr_(&CONSTL, &CONSTT, &CONST6, &CONST302, &CONST3, A, &CONST6, TAU,
B, &CONST6, WORKB, &CONST302, &INFO);

//the multiplication of the matrix A' and the matrix B results in a matrix
// of dimensions 3x302 but is stored in therefore it is migrated to M
for(j=0;j<302;j++)
		for(k=0;k<3;k++)
				M[j*3+k] = B[j*6+k];

//use clapack to solve R * M = (Q' * B) for M (the dipole moment) where R is the upper triangular
//matrix generated by dgeqp3
//and the value Q'*B is the result of dormqr which is now contained in M
//the result overwrites M
dtrtrs_(&CONSTU, &CONSTN, &CONSTN, &CONST3, &CONST302, A, &CONST6, M,
&CONST3, &INFO);

/*multiple the result M by the previously generated pivot matrix to align columns with actual
solution*/
//the result which is the dipole moment is contained in matrix L
dgemm_(&CONSTN, &CONSTN, &CONST3, &CONST302, &CONST3, &ALPHA, PIVOT,
&CONST3, M, &CONST3, &BETA, L, &CONST3);

//attempt to reconstruct original signal by multiplying the dipole moment L with the lead vector C
//resultant reconstructed EKG signal is stored into NEW.
dgemm_(&CONSTN, &CONSTN, &CONST6, &CONST302, &CONST3, &ALPHA, C,
&CONST6, L, &CONST3, &BETA, NEW, &CONST6);

//find Maximum Absolute Value of each row of the Normal Patient
if(MaxAbs[0] == 0){		//ensures that for each reconstruction the normalization occurs once.
		for(i=6;i<12;i++){
			hold2=0;
			for(j=0;j<302;j++){
				if(pat[j*12+i] < 0){
					hold1 = pat[j*12+i] * -1;
				}
				else
					hold1 = pat[j*12+i];
				hold1 = fabs(hold1);
				if(hold1 > hold2){
					hold2 = hold1;
				}

			}
			MaxAbs[i-6] = hold2;
		}
```

```
                        //Normalize normal patient vectors for Normal Patient
                        for(i=6;i<12;i++){
                                for(j=0;j<302;j++){
                                        NormXact[i-6][j] = pat[j*12+i] / MaxAbs[i-6];
                                }
                        }
                }
                //Normalize current reconstructed vector using saved results from original patient normalization
                //also compute RMS error for each of the 6 vectors
                //sum all six RMS errors into SumErr which represents the total error for the reconstruction
                SumErr = 0;
                for(i=0;i<6;i++){
                        RMSerr[i] = 0;
                        for(j=0;j<302;j++){
                                hold1 = NormXact[i][j] - (NEW[j*6+i] / MaxAbs[i]);
                                hold1 = hold1 * hold1;
                                RMSerr[i] = RMSerr[i] + hold1;
                        }
                        RMSerr[i] = sqrt(RMSerr[i]/302);
                        SumErr += RMSerr[i];
                }
                //determine if error for current reconstruction is less that previous errors.
                //if so save error and index as possible solution
                if(SumErr < MinErr){
                        MinErr = SumErr;
                        MinInd = index;
                }

        }
        //printf("Min Error = %9.20f at Index = %d\n", MinErr, MinInd);

        //Store min error and index into struct
        result.error = MinErr;
        result.index = MinInd;

        return (&result);
}

void open_files(){
        int fd, err;

        //printf("\n\nOPENING THE FILES\n\n");

        //open file and check for errors
        fd = open("../data/LeadVectorC.d", O_RDONLY);
        if(fd < 0){
                printf("Error %i opening file ../data/LeadVectorC.d", fd);
                return;
        }
        //read file and check for errors
        err = read(fd, &matrix, sizeof(matrix));
        if(err <= 0){
                printf("Error %i reading file ../data/LeadVectorC.d", err);
                return;
        }
        close(fd); //close file
        //open Patient file and check for errors
        fd = open("../data/NormalPatientC.d", O_RDONLY);
        if(fd <= 0){
                printf("Error %d opening ../data/NormalPatientC.d\n", fd);
                return;
        }
```

```
	//Read Patient File
	err = read(fd, &patient, sizeof(patient));
	if(err<=0){
		printf("Error %d reading ../data/NormalPatientC.d\n", fd);
		return;
	}
	close(fd);
}
```

A.3.5 Threaded Server File (*server.PTH.c)*

The code used in the server threaded implementation is shown below.

```
#include "Reconstruct.h"
#include <string.h>
#include "f2c.h"
#include "blaswrap.h"
#include <fcntl.h>
#include <math.h>

pthread_mutex_t mutex;
pthread_mutex_t mutex_lead;
pthread_mutex_t mutex_pat;
double matrix[3][20571][14];		//buffer for lead vector matrix
double patient[12][302];		//buffer for patient matrix
double MinErr = 99999;
int    MinInd;
int bool_open = 0;
double *mat, *pat;
double *B_hold;
double MaxAbs[6] = {0, 0,0,0,0,0};
double NormXact[6][302];

struct thread_area{
	int start_index;
	int end_index;
	double *A;
	double *B;
	double *C;
	double *L;
	double *M;
	double *NEW;
	double *PIVOT;
	double *TAU;
	double *WORK;
	double *WORKB;
} thread_args[8];

void compute(void *);
void open_files();

rec_out * qrerror_1_svc(rec_in *argsptr, struct svc_req *rqstp){
	int thread_count, diff, i, j, increment, err, count;
	static rec_out result;
	pthread_t threads[8];


	if(bool_open == 0){
		bool_open = 1;
		open_files();
```

```
	}

	//printf("\nThe starting index = %d and the ending index = %d\n", argsptr->start_index, argsptr->end_index);

	mat = (double *) malloc(sizeof(matrix));
	pat = (double *) malloc(sizeof(patient));
	B_hold = (double *) malloc(6*302*sizeof(double));
	//use pointers to access lead vector and original patient data.
	mat = (double *) matrix;
	pat = (double *) patient;
	//copy patient data into HOLD matrix to be used later to refresh patient data
	for(i=0;i<302;i++)
		for(j=0;j<6;j++)
			B_hold[i*6+j] = pat[i*12+j+6];

	MinErr = 9;		//initialize error
	//calculate increment for indexes
	diff = argsptr->end_index - argsptr->start_index;
	increment = diff/(8);
	if(diff%8 != 0)
		increment++;
	for(thread_count = 0; thread_count < 8; thread_count++){
		//calculate starting and ending indexes for threads
		if(thread_count == 0)
			thread_args[thread_count].start_index = argsptr->start_index;
		else
			thread_args[thread_count].start_index = increment * thread_count + argsptr->start_index;
		if(thread_count == 7)
			thread_args[thread_count].end_index = argsptr->end_index;
		else
			thread_args[thread_count].end_index = thread_args[thread_count].start_index +
			increment;
		//create thread
		if((err = pthread_create(&threads[thread_count], NULL, (void *) compute,
		&thread_args[thread_count])) != 0){
			printf("Error creating thread #%d\n", thread_count+1);
		}
	}
	//wait for threads
	for(thread_count = 0; thread_count < 8; thread_count++){
		pthread_join(threads[thread_count],NULL);
	}

	//Store min error and index into struct
	result.error = MinErr;
	result.index = MinInd;

	return (&result);
}

void open_files(){
	int fd, err;

	//printf("\n\nOPENING THE FILES\n\n");

	//open file and check for errors
	fd = open("../data/LeadVectorC.d", O_RDONLY);
	if(fd < 0){
		printf("Error %i opening file ../data/LeadVectorC.d", fd);
		return;
	}
```

```
	//read file and check for errors
	err = read(fd, &matrix, sizeof(matrix));
	if(err <= 0){
		printf("Error %i reading file ../data/LeadVectorC.d", err);
		return;
	}
	close(fd); //close file
	//open Patient file and check for errors
	fd = open("../data/NormalPatientC.d", O_RDONLY);
	if(fd <= 0){
		printf("Error %d opening ../data/NormalPatientC.d\n", fd);
		return;
	}
	//Read Patient File
	err = read(fd, &patient, sizeof(patient));
	if(err<=0){
		printf("Error %d reading ../data/NormalPatientC.d\n", fd);
		return;
	}
	close(fd);
	return;
}

void compute(void *threadargs){

	struct thread_area *args;
	int index;
	int count;
	int i;
	int j;
	int k;
	integer INFO;
	integer JPVT[3];
	integer CONST3 = 3,
		CONST6 = 6,
		CONST302 = 302;
	char CONSTN = 'N',
		CONSTL = 'L',
		CONSTT = 'T',
		CONSTU = 'U';
	double ALPHA = 1.0, BETA = 0.0;
	double hold1;
	double hold2;
	double NormApp[6][303];
	double Error[6][302];
	double RMSerr[6];
	double SumErr;
	double MinErrLoc = 999;
	int MinIndLoc;

	args = (struct thread_area *)threadargs;

	//dynamic memory allocation per thread
	args->A = (double *) malloc(6*3*sizeof(double));
	args->B = (double *) malloc(6*302*sizeof(double));
	args->C = (double *) malloc(6*3*sizeof(double));
	args->L = (double *) malloc(3*302*sizeof(double));
	args->M = (double *) malloc(3*302*sizeof(double));
	args->NEW = (double *) malloc(CONST6*CONST302*sizeof(double));
	args->PIVOT = (double *) malloc(3*3*sizeof(double));
	args->TAU = (double *) malloc(3*sizeof(double));
	args->WORK = (double *) malloc(CONST302*sizeof(double));
```

```
args->WORKB = (double *) malloc(CONST302*sizeof(double));


for(index = args->start_index;index < args->end_index; index++){
        count = 0;
        pthread_mutex_lock(&mutex_lead);
        for(i=0;i<3;i++){
                for(j=0;j<6;j++){
                        //store lead vector data into matrix A to preserve integrity of data
                        args->A[count] = mat[(i*287994)+(j*20571)+index];
                        args->C[count] = mat[(i*287994)+(j*20571)+index];
                        count++;
                }
        }
        pthread_mutex_unlock(&mutex_lead);
        //initialize JPVT pointer
        for(i=0;i<3;i++)
                JPVT[i] = 0;
        pthread_mutex_lock(&mutex_pat);
        memcpy(args->B, B_hold, 6*302*sizeof(double));          //refresh original patient matrix B
        pthread_mutex_unlock(&mutex_pat);

        /* call clapack routine to obtain QR factorization of A. A is overwritten with orthogonal matrix Q
        and triangular matrix R*/
        pthread_mutex_lock(&mutex);
        dgeqp3_(&CONST6, &CONST3, args->A, &CONST6, &JPVT, args->TAU, args->WORK,
        &CONST302, &INFO);

        /*generate pivot matrix later used to obtain dipole moment as part of the equation A*P = Q*R
        where P is the pivot matrix*/
        for(i=0;i<3;i++)
                for(j=0;j<3;j++)
                        if(j==JPVT[i]-1)
                                args->PIVOT[i*3+j] = 1.0;
                         else
                                args->PIVOT[i*3+j] = 0.0;

        //use clapack routine to multiple orthogonal matrix Q' contained in A by lead vector matrix B
        //B is overwritten with product
        dormqr_(&CONSTL, &CONSTT, &CONST6, &CONST302, &CONST3, args->A, &CONST6,
        args->TAU, args->B, &CONST6, args->WORKB, &CONST302, &INFO);

        //the multiplication of the matrix A' and the matrix B results in a matrix
        // of dimensions 3x302 but is stored in B therefore it is migrated to M which is 3x302
        for(j=0;j<302;j++)
                for(k=0;k<3;k++)
                                args->M[j*3+k] = args->B[j*6+k];

        /*use clapack to solve R * M = (Q' * B) for M (the dipole moment) where R is the upper triangular
        matrix generated by dgeqp3*/
        //and the value Q'*B is the result of dormqr which is now contained in M
        //the result overwrites M
        dtrtrs_(&CONSTU, &CONSTN, &CONSTN, &CONST3, &CONST302, args->A, &CONST6,
        args->M, &CONST3, &INFO);

        /*multiple the result M by the previously generated pivot matrix to align columns with actual
        solution*/
        //the result which is the dipole moment is contained in matrix L
        dgemm_(&CONSTN, &CONSTN, &CONST3, &CONST302, &CONST3, &ALPHA, args-
        >PIVOT, &CONST3, args->M, &CONST3, &BETA, args->L, &CONST3);

        //attempt to reconstruct original signal by multiplying the dipole moment L with the lead vector C
```

```
		//resultant reconstructed EKG signal is stored into NEW.
		dgemm_(&CONSTN, &CONSTN, &CONST6, &CONST302, &CONST3, &ALPHA, args->C,
		&CONST6, args->L, &CONST3, &BETA, args->NEW, &CONST6);
		pthread_mutex_unlock(&mutex);

		//find Maximum Absolute Value of each row of the Normal Patient
		if(MaxAbs[0] == 0){			//ensures that for each reconstruction once.
			for(i=6;i<12;i++){
				hold2=0;
				for(j=0;j<302;j++){
					if(pat[j*12+i] < 0){
						hold1 = pat[j*12+i] * -1;
					}
					else
						hold1 = pat[j*12+i];
					hold1 = fabs(hold1);
					if(hold1 > hold2){
						hold2 = hold1;
					}

				}
				MaxAbs[i-6] = hold2;
			}
			//Normalize normal patient vectors for Normal Patient
			for(i=6;i<12;i++){
				for(j=0;j<302;j++){
					NormXact[i-6][j] = pat[j*12+i] / MaxAbs[i-6];
				}
			}
		}
		//Normalize current reconstructed vector using saved results from original patient normalization
		//also compute RMS error for each of the 6 vectors
		//sum all six RMS errors into SumErr which represents the total error for the reconstruction
		SumErr = 0;
		for(i=0;i<6;i++){
			RMSerr[i] = 0;
			for(j=0;j<302;j++){
				hold1 = NormXact[i][j] - (args->NEW[j*6+i] / MaxAbs[i]);
				hold1 = hold1 * hold1;
				RMSerr[i] = RMSerr[i] + hold1;
			}
			RMSerr[i] = sqrt(RMSerr[i]/302);
			SumErr += RMSerr[i];
		}
		//determine if error for current reconstruction is less that previous errors.
		//if so save error and index as possible solution
		if(SumErr < MinErrLoc){
			MinErrLoc = SumErr;
			MinIndLoc = index;
		}
	}
	if(MinErrLoc < MinErr){
		MinErr = MinErrLoc;
		MinInd = MinIndLoc;
	}
	return;
}
```

Appendix B

Tables

This section includes tables with all of the values obtained from the different test runs on different numbers of nodes. These values were used to obtain the averages presented in the body of this thesis.

Table 10: Run Times - 1 Node

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	1552231	1541724	1625033
2	1554668	1552036	1595178
3	1548872	1546351	1648996
4	1556206	1557091	1602862
5	1549227	1548820	1614258
6	1560973	1543692	1627045
7	1550903	1549188	1607450
8	1555837	1542931	1622967
9	1551566	1556310	1641310
10	1554237	1550502	1621277

Table 11: Run Times - 3 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	527839	523800	548670
2	518383	523626	589328
3	523486	525450	545392
4	525065	527406	547213
5	523619	525121	554034
6	526076	526680	545838
7	523401	522487	594533
8	526655	521740	542221
9	523709	522534	551563
10	525872	523767	543681

Table 12: Run Times - 2 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	782514	782300	817129
2	783602	780668	821816
3	777278	783540	814153
4	782225	779972	820099
5	783420	783082	818771
6	782406	777597	813924
7	782348	786938	807688
8	782309	777944	821040
9	785145	786698	819817
10	781409	781042	812892

Table 13: Run Times - 4 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	396268	395318	417235
2	397861	394530	418618
3	394705	396566	410525
4	397327	394316	412951
5	397067	395957	442904
6	398637	395789	408771
7	395362	393865	427715
8	397434	397651	420739
9	399216	396886	412532
10	397293	396736	415734

Table 14: Run Times - 5 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	319916	318106	333358
2	319075	320291	345388
3	320266	316794	350389
4	322470	317848	333478
5	321102	319420	333835
6	319398	318495	330361
7	318740	319977	349924
8	316802	319784	339231
9	320622	316852	338110
10	319772	317548	333495

Table 15: Run Times - 7 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	233105	230731	258925
2	233615	232213	248566
3	231417	230297	240086
4	232881	231771	240729
5	232891	231663	244506
6	232862	229988	245113
7	231081	231288	244603
8	232738	234963	239360
9	233979	230127	241839
10	233202	230707	247954

Table 16: Run Times - 6 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	269217	268622	276272
2	267892	267469	289357
3	266753	268043	281473
4	270820	265763	275800
5	270916	268194	274961
6	270375	268646	287871
7	267716	268346	287851
8	267880	265291	278265
9	269746	267224	280905
10	269486	266869	280325

Table 17: Run Times - 8 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	207542	203566	217811
2	205789	202307	217070
3	206283	203491	209691
4	204565	202397	227640
5	204491	204132	212875
6	205278	202746	220854
7	206298	204006	212010
8	204855	204240	218278
9	205369	204554	214547
10	206654	205608	211665

Table 18: Run Times - 9 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	186894	182113	205422
2	183307	182872	207833
3	184000	182482	209227
4	183790	181849	194113
5	186185	179911	192155
6	183680	180744	187465
7	185394	180547	188273
8	184508	181506	219194
9	184392	182911	189991
10	184383	182711	192934

Table 19: Run Times - 11 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	155222	152140	160372
2	154262	150361	164239
3	153679	150787	167025
4	153677	151362	157216
5	154864	150879	160499
6	154152	151484	158899
7	154895	151502	158064
8	154075	152231	164315
9	155580	151513	154845
10	156655	150813	160633

Table 20: Run Times - 10 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	169726	165096	170495
2	169622	166267	171138
3	168808	164346	169339
4	167066	165680	177897
5	168398	165109	174158
6	167880	165873	174478
7	168539	165337	182273
8	169055	165465	169382
9	168225	165051	176169
10	167270	165862	176191

Table 21: Run Times - 12 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	142844	139522	158205
2	143154	141411	151739
3	143305	141152	170011
4	143780	140218	146673
5	142732	141191	143022
6	145009	140335	144176
7	143624	141502	143776
8	143001	139737	146149
9	143458	139621	150173
10	142308	140812	146742

Table 22: Run Times - 13 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	134562	130688	139396
2	134997	130064	137072
3	134165	130598	139949
4	134646	129857	136235
5	134005	129364	135908
6	133698	131155	137932
7	134715	129602	140679
8	134862	130264	138647
9	134994	130690	133503
10	133722	131023	135574

Table 23: Run Times - 15 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	119255	115887	117073
2	120824	114743	138131
3	118474	115165	115941
4	119706	115502	124504
5	118440	115379	133040
6	120717	115389	122976
7	117346	115278	126848
8	120345	115895	119682
9	120231	114797	116368
10	118544	114405	122202

Table 24: Run Times - 14 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	127004	123166	132463
2	126018	121515	125312
3	126092	122177	130737
4	126855	121729	123557
5	126000	121762	124615
6	124808	121126	130724
7	126623	121375	133960
8	127897	121612	123870
9	126344	121938	132362
10	126934	120810	140030

Table 25: Run Times - 16 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	113850	108936	115605
2	113112	109038	120395
3	112666	108314	116823
4	113029	108528	120341
5	113847	108563	114807
6	114165	108945	119874
7	115123	107671	114741
8	113453	108466	112262
9	113018	108930	126194
10	113147	108662	114140

Table 26: Run Times - 17 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	108080	103597	110162
2	107709	102701	120200
3	109123	102837	114753
4	108998	103892	105863
5	106846	102821	117150
6	109404	102915	107646
7	107533	103730	109446
8	107613	104196	111412
9	108904	103000	107874
10	109518	103840	120093

Table 27: Run Times - 19 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	100710	93142	103665
2	98640	94963	101713
3	99096	93231	106563
4	100226	94452	106094
5	100361	94797	104979
6	99728	93941	103508
7	98154	94408	99659
8	97839	94238	99732
9	101163	94050	108956
10	99233	94527	104299

Table 28: Run Times - 18 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	104755	98714	103951
2	103747	98336	104361
3	103788	98579	105318
4	102537	98668	106464
5	104776	98026	101862
6	103474	98035	114060
7	103330	97461	102852
8	103636	98700	115464
9	103571	98018	106552
10	103673	98945	97482

Table 29: Run Times - 20 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	95021	90268	94384
2	97403	89632	90559
3	97267	89885	97801
4	95999	89783	92474
5	94753	90402	93259
6	95231	90619	95960
7	96448	90606	96383
8	96473	89358	102397
9	98527	90535	102170
10	95912	91079	90607

Table 30: Run Times - 21 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	92181	86346	93014
2	93054	86948	87739
3	93149	87325	93101
4	93962	87446	93731
5	93844	85883	90831
6	93235	87766	87786
7	92891	87494	98994
8	91434	86191	90950
9	95004	87381	92884
10	91304	87804	90047

Table 31: Run Times - 23 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	85581	81778	83077
2	88258	79716	89273
3	87447	81180	88988
4	87113	80667	85281
5	86245	80956	85263
6	87654	80401	81960
7	86765	80125	82761
8	88363	81170	80679
9	85252	81351	87154
10	88806	81480	82969

Table 32: Run Times - 22 Nodes

	Distributed Implementation (us)		
Run	**Fork**	**Thread**	**Thread2**
1	87697	84306	90911
2	87285	82396	90137
3	89574	83295	85884
4	89994	83009	88155
5	91248	83922	90915
6	88946	83334	88661
7	90197	83162	92565
8	89843	83957	89817
9	88845	84233	91735
10	89402	84539	94116

B.1 Memory Allocation Times

Below is the table that contains the different times taken to dynamically allocate the memory necessary for the reconstruction.

Table 33: Memory Allocation Times

Run	Time(us)	Run	Time(us)
1	16.375	**24**	170.75
2	15.875	**25**	17.625
3	15	**26**	17.5
4	14.125	**27**	191.875
5	53.5	**28**	26
6	283.375	**29**	17.5
7	15	**30**	281.625
8	14.35	**31**	262.375
9	95.125	**32**	27.625
10	15.25	**33**	427.875
11	16.625	**34**	19.125
12	33.3	**35**	437.375
13	23.375	**36**	68
14	15.75	**37**	25
15	16.25	**38**	22
16	16.375	**39**	62
17	15.25	**40**	19.75
18	18.5	**41**	19.25
19	16.125	**42**	23.25
20	17	**43**	133.25
21	16.625	**44**	215.25
22	17.25	**45**	352.5
23	17.625	**46**	28.125

Curriculum Vitae

Gabriel Cordova was born on July 3, 1984, in Bellflower, CA. He was the younger of two children brought up by their single mother, Gabriela Palma. Gabriel graduate from Coronado High School at El Paso, TX in May of 2002. He then attended the University of Texas at El Paso where he obtained a Bachelor of Science degree in electrical engineering with an emphasis on computer engineering. While completing his undergraduate studies Gabriel worked part time as a programmer for Currey Adkins, an information technology company in El Paso, TX. Upon obtaining his Bachelor degree in December 2006, Gabriel continued his education at the University of Texas at El Paso by pursuing a Master of Science degree in computer engineering. During his graduate studies Gabriel maintained a research position where he focused on distributed computing, and served as a system administrator for the UNIX Lab in the Electrical and Computer Engineering Department of UTEP. In the summer of 2007 he completed an internship with Hewlett-Packard in Fort Collins, CO as part of the Firmware System Test Lab. Gabriel completed his graduate studies in December 2008, and accepted a full-time position with Hewlett-Packard as part of the Systems VLSI Lab.

Permanent Address: 7732 Iroquois Dr.

El Paso, TX 79912

This thesis was typed by Gabriel Cordova.