

**PARALLELIZATION OF ECG TEMPLATE-BASED
ABNORMALITY DETECTION**

By

Sherry L. Kratsas

**A THESIS
Submitted to
The College of Engineering
and Mineral Resources
at
West Virginia University
in partial fulfillment of the requirements
for the degree of
Master of Science
Electrical Engineering**

**Bojan Cukic, Ph.D., Chair
Stephanie Schuckers, Ph.D.
Wils Cooley, Ph.D.**

Department of Computer Science and Electrical Engineering

**Morgantown, West Virginia
2000**

**Keywords: Parallel Algorithms, Beowulf Cluster, Electrocardiogram Processing,
Abnormality Detection**

ABSTRACT

PARALLELIZATION OF ECG TEMPLATE-BASED ABNORMALITY DETECTION

By Sherry L. Kratsas

Approximately 225,000 adults experience sudden cardiac death each year in the United States. In order to develop techniques to predict cardiac arrest, long-term study of electrocardiogram (ECG) data needs to be done to detect changes in electrical activity of diseased hearts. The goal of such study is to determine a set of electrical patterns that indicate the deterioration of the heart and possibility of cardiac arrest.

In the past, limitations of computing power and storage space restricted the duration of long-term studies to several days. However, with today's technological advancement, data collection can be extended to months or years. The goal of this thesis research is to evaluate several alternatives for distributing the analysis of ECG data over multiple processors. Parallel algorithms utilizing Correlation Waveform Analysis (CWA) were implemented to compare individual heartbeats and form heartbeat templates. The purpose of the templates is to exhibit the different heartbeat morphologies encountered in the data. The processing is done on a Linux Beowulf Cluster using the standardized Message Passing Interface (MPI) libraries. In the thesis, the results of four different parallel approaches are compared, and their performance is evaluated.

Table of Contents

List of Figures	iv
List of Tables.....	vi
List of Abbreviations.....	vii
Chapter 1 Introduction	1
Chapter 2 Related Work.....	4
2.1 Implantable Cardioverter Defibrillator Research.....	4
2.2 Morphology Analysis with Low Computational Demands	6
2.3 Long-Term Electrocardiogram Collection.....	9
2.4 Thesis Foundation.....	11
2.5 Parallel Algorithms	19
2.6 Message Passing Libraries.....	20
Chapter 3 Parallel Algorithms for Template-Based ECG Processing.....	25
3.1 Root-Only Template Formation (ROTF) Algorithm	27
3.2 Multiple Process Correlation Computation (MPCC) Algorithm.....	30
3.3 Node Data Server (NDS) Algorithm.....	34
3.4 Thread Data Server (TDS) Algorithm	37
Chapter 4 Results	41
4.1 Time and Efficiency Analysis.....	41
4.1.1 ROTF Results	42
4.1.2 MPCC Results	45
4.1.3 NDS Results	46
4.1.4 TDS Results.....	48
4.2 Comparison of Methods.....	51
4.3 Templates Generated	54
Chapter 5 Summary.....	57
5.1 Review	57
5.2 Future Work.....	58
5.3 Lessons Learned.....	60
References	61

List of Figures

<u>Label</u>	<u>Description</u>	<u>Page</u>
Figure 1	ECG with baseline drifting before bandpass filter is applied.	13
Figure 2	ECG after bandpass filter is applied.	13
Figure 3	General Overview of ECG Template Generation	18
Figure 4	Block Diagram of Root-Only Template Formation (ROTF) Algorithm	29
Figure 5	Timeline of Root-Only Template Formation (ROTF) Algorithm	30
Figure 6	Block Diagram of Multiple Process Correlation Computation (MPCC) Algorithm	32
Figure 7	Timeline of Multiple Process Correlation Computation (MPCC) Algorithm	33
Figure 8	Block Diagram of Node Data Server (NDS) and Thread Data Server (TDS) Algorithms	39
Figure 9	Timeline of Node Data Server (NDS) and Thread Data Server (TDS) Algorithms	40
Figure 10	Results for Root-Only Template Formation (ROTF) Algorithm	44
Figure 11	Results for Multiple Process Correlation Computation (MPCC) Algorithm	46
Figure 12	Results for Node Data Server (NDS) Algorithm	48
Figure 13	Results for Thread Data Server (TDS) Algorithm	50
Figure 14	Comparison of Algorithms (Results from Processing Twelve Files)	51
Figure 15	Comparison of ROTF and NDS (Results from Processing Twelve Files)	52
Figure 16	Normal Heartbeat vs. Abnormal Template	55

Figure 17	Example of a template that may be noise or a cardiac event.	55
Figure 18	Example of a template formed from a noisy signal.	56

List of Tables

<u>Label</u>	<u>Description</u>	<u>Page</u>
Table 1	Instruction Definitions for Node Data Server (NDS) and Thread Data Server (TDS) Algorithms	35
Table 2	Data Collected using the Root-Only Template Formation (ROTF) Algorithm (Results from Processing Six Files)	43
Table 3	Data Collected using the Multiple Process Correlation Computation (MPCC) Algorithm (Results from Processing Six Files)	45
Table 4	Data Collected using the Node Data Server (NDS) Algorithm (Results from Processing Six Files)	47
Table 5	Data Collected using the Thread Data Server (TDS) Algorithm (Results from Processing Six Files)	49

List of Abbreviations

<u>Abbreviation</u>	<u>Meaning</u>
ADIOS	Accumulated Difference of Slopes
BAM	Bin Area Method
CWA	Correlation Waveform Analysis
DAM	Derivative Area Method
ECG	Electrocardiogram
ICD	Implantable Cardioverter Defibrillator
MPCC	Multiple Process Correlation Computation
MPI	Message Passing Interface
NAD	Normalized Area of Difference
NDS	Node Data Server
NFS	Network File System
ROTF	Root-Only Template Formation
SCD	Sudden Cardiac Death
TDS	Thread Data Server
VF	Ventricular Fibrillation
VT	Ventricular Tachycardia

Chapter 1

Introduction

In the United States, approximately 225,000 adults experience sudden cardiac death (SCD) each year. Extensive research has been performed in recent decades to develop implantable devices that can detect and treat cardiac arrest in order to save lives. These devices are commonly used today. Further research is being done to expand the capabilities of these devices to allow them to predict cardiac arrest, so that patients and doctors may take preventative measures to avoid potentially fatal cardiac events.

Long-term collection and study of electrocardiogram (ECG) data is an essential asset in developing techniques to predict cardiac problems. The continuous monitoring of ECGs accumulates massive amounts of data. In the past, the duration of study was restricted to several days due to limited storage capacity. The amount of data collected in several days is not sufficient to develop an understanding of the changes that can occur as heart disease develops. Today's technological advancement and affordability of data storage resources allow the ECG collection effort to be extended to months or years.

The source of the data for this research is an experiment in collection of continuous ECGs from rabbits. The data samples are collected at 1000 samples per second for several weeks. The size of each sample is 2 bytes. One day's worth of data for one animal becomes approximately 170 Mbytes. The eventual goal of the experiment is to monitor the rabbits for 4 weeks as control, and then gradually inject them with a drug to induce heart disease for an additional 10 weeks. If four animals were monitored for 4 months, the amount of data accumulated would exceed 81 Gbytes. Furthermore, if

two channels, one for surface and one for intracardiac collection, were monitored for each animal, the amount of data would double. One Pentium III 500 MHz personal computer typically takes one to one and a half minutes to process one 8 Mbyte file. Therefore, the time necessary to process 81 Gbytes of data on one personal computer would be approximately 11 days. Processing times this high are inconvenient for researchers.

There is a need for "super-computing" capabilities to process the amount of data accumulated from long-term, continuous ECG collection. The goal of the research described herein is to evaluate a few alternatives for distributing the processing of large ECG data sets over multiple processors in a Beowulf cluster. A Beowulf cluster is simply a group of standard personal computers connected through a local area network running a Unix-based operating system. Clusters possess distributed memory, but they utilize standard message-passing libraries to simulate a shared memory environment. Therefore, a group of machines can work together as a single virtual machine. Parallel programs can be written to use the clusters as cheap supercomputers. Clusters are ideal in research applications. They are much less expensive than multiprocessor systems offering comparable advantages. Clusters are scalable and the computers can easily be reused in ordinary applications.

The Beowulf cluster used in this research is running RedHat Linux 6.1 and utilizing the standardized MPI library, version 1.2.0. The cluster is composed of six 500 MHz Pentium III personal computers and a Cisco Catalyst 2900 series XL 10BaseT/100BaseTX switch.

In Chapter 2, an overview of research related to one medical device, the implantable cardioverter defibrillator (ICD), is given. The problems of ICD devices and several techniques for improving the devices' capabilities are discussed. In addition, some cases of long-term ECG collection are presented. Chapter 2 also describes the existing software that implements abnormality detection in ECG data. The third chapter discusses four ways in which the abnormality detection algorithm is distributed among the multiple processors of a cluster. The results and performance of the different parallel implementations are compared in Chapter 4. A summary and discussion of future work are presented in Chapter 5.

Chapter 2

Related Work

2.1 Implantable Cardioverter Defibrillator Research

Numerous efforts have been made in past years to develop treatments and preventative measures to aid in the battle against SCD. One particular effort, the implantable cardioverter defibrillator (ICD), is the source of motivation for this thesis research. When implanted into a patient, an ICD monitors the electrical activity of the heart. When it detects threatening electrical activity such as ventricular tachycardia (VT) or ventricular fibrillation (VF), the ICD administers an electrical shock to return a normal rhythm to the heart. As of 1995, over 75,000 ICDs had been implanted into people and that number has significantly increased since then [1]. The device has saved thousands of lives, but it still has room for improvement.

Many aspects of ICD design are under review for improvement. The devices have been known to deliver false electrical shocks to patients when they are not needed. These false shocks cause several problems. The first, and most obvious, problem is the unnecessary discomfort the shocks cause for patients. Secondly, unnecessary shocks use battery power. As a result, the device is left with less power for legitimate treatment. In addition, consuming power reduces the time until the device needs to be removed or recharged. The most dangerous effect of a false shock is the possible initiation of VT or VF [1]. SCD, by definition, results from the abrupt loss of heart function known as

cardiac arrest [2]. In most cases, VT or VF initiates cardiac arrest. The ICD was developed to detect abnormal electrical activity such as VT and VF in ECGs. Therefore, the initiation of VT or VF by the devices completely defeats their purpose.

In order to eliminate the possibility of false shocks, new signal processing algorithms have been investigated for their potential in successfully detecting VT and VF. In early ICDs, most detection algorithms simply monitored heart rates. Later efforts involved the incorporation of morphology analysis [3]. When implementing morphology analysis, computational demands of algorithms became an important consideration. The earliest morphology difference detection was achieved with the probability density function (PDF) [4]. PDF was fairly reliable in detecting VF, but it was not as reliable at VT detection. Correlation Waveform Analysis (CWA) was found to be highly successful in detecting VT [5]. When using CWA, a correlation coefficient, ρ , is calculated by Equation 1.

$$\rho = \frac{\sum_{i=1}^{i=N} (ti - \bar{t})(si - \bar{s})}{\sqrt{\sum_{i=1}^{i=N} (ti - \bar{t})^2 \sum_{i=1}^{i=N} (si - \bar{s})^2}}$$

Equation 1: Calculation of Correlation Coefficient using Correlation Waveform Analysis (CWA)

This calculation gives a result between -1 and $+1$ in which $+1$ signifies a perfect match between the template and the signal under analysis. In the equation, N is the number of points in the templates being compared, ti is a single template sample, \bar{t} is the average of all the samples in the template, si is a single sample of the signal being analyzed, and \bar{s} is the average of all the samples in the signal being analyzed [5].

One advantage of using CWA is that it is independent of amplitude or baseline fluctuations unlike previous algorithms for morphology analysis had been. The disadvantage of using CWA is its high computational demands. This makes it unattractive for incorporation in an implantable device. This problem led to the investigation of the VT detection potential of other less computationally demanding algorithms.

CWA is used in the parallel algorithms presented in this paper. The goal of this research is not to decrease processing time of a morphology analysis algorithm for use in an ICD. The objective is to decrease the processing time for the analysis of large data sets that are collected previous to processing. The accuracy of the method in detecting various morphologies is more important than its computational demands.

2.2 Morphology Analysis with Low Computational Demands

In one study, four time-domain methods were developed for comparison with CWA. The first algorithm proposed was the Bin Area Method (BAM) [3]. This method, like CWA, processes the signal directly and returns a measurement between -1 and $+1$. BAM is also independent of amplitude and baseline fluctuations. It forms a set of bins from the samples of a waveform or template. The bins are formed by adding consecutive samples and the size of these bins can vary. For example, if five-point bins are used, $S1 = s1 + s2 + s3 + s4 + s5$, $S2 = s6 + s7 + s8 + s9 + s10$, and $SM = SN-4 + SN-3 + SN-2 + SN-1 + SN$. The bins serve to apply the rectangular area rule for approximating the area under a signal. The coefficient value is calculated by Equation 2.

$$\rho = 1 - \sum_{i=1}^M \left| \frac{Ti - \bar{T}}{\sum_{k=1}^M |Tk - \bar{T}|} - \frac{Si - \bar{S}}{\sum_{i=1}^M |Sk - \bar{S}|} \right|$$

Equation 2: Calculation of Correlation Coefficient using Bin Area Method (BAM)

S_i and S_k are the bin values for the signal being compared, \bar{S} is the average of the bin values for the waveform, T_i and T_k are the bin values for the template, \bar{T} is the average of the bin values in the template, and M is the number of bins. This algorithm replaces the $2N + 2$ multiplications of CWA, with $N/X + 1$ multiplications, where X is the size of the bins [3].

The second algorithm developed also dealt with the signal directly. This algorithm, called Normalized Area of Difference (NAD), is identical to BAM except that it does not subtract the averages of the bin values [3]. This makes the algorithm vulnerable to baseline fluctuations. It is still independent of amplitude changes. NAD replaces the $2N + 2$ multiplications and one division of CWA with $N/X + 1$ multiplications and zero divisions.

The third method does not deal with the signal alone. The name of the method is the Derivative Area Method (DAM) [3]. It uses the derivative of the signal for comparison. A "zero crossing array" (ZCA) is built to partition the template into segments. The ZCA is computed by comparing a sample to the previous sample. If one derivative sample has the same sign as the previous sample, a 0 is placed into the ZCA. If the sign differs from the previous sample, a 1 is entered into the array. Once the template is partitioned in this fashion, the same partition is applied to waveform that is

compared to the template. The area under the template in each partition is calculated and compared to the corresponding values of the signal. The comparing coefficient is computed by Equation 3.

$$\rho = 1 - \sum_{k=1}^M \left| \frac{Tk'}{\sum_{j=1}^M |Tj'|} - \frac{Sk'}{\sum_{j=1}^M |Sj'|} \right|$$

Equation 3: Calculation of Correlation Coefficient using Derivative Area Method (DAM)

In this calculation, Tk' and Sk' are the derivative samples of the signal and M is the number of partitions. This method results in M number of multiplications [3]. The equation also returns a value between -1 and $+1$.

The last algorithm developed in the study was the Accumulated Difference of Slopes (ADIOS) [3]. This method also utilizes the derivative of the signal under analysis. This method simply compares the sign of the derivative sample of the template with the sign of the derivative being processed by using the logical XOR (\oplus) operation. The value of -1 to $+1$ is calculated by Equation 4.

$$\rho = \sum_{i=1, N} \text{sign } ti' \oplus \text{sign } si'$$

Equation 4: Calculation of Correlation Coefficient using Accumulated Difference of Slopes (ADIOS)

All of these algorithms were compared to CWA using a data set from patients participating in a study in which the occurrences of VT were known and examined. One

algorithm, NAD, produced better results in detecting VT than CWA. However, this algorithm is also the one that is vulnerable to baseline fluctuations.

2.3 Long-Term Electrocardiogram Collection

Research aimed at the improvement of the ICD is currently being done at the Biomedical Signal Analysis Laboratory at West Virginia University. Part of the research is focused on searching for electrical events that occur in an ECG that could indicate that a person is in danger of future cardiac arrest. Experiments are being conducted in which ECG data is collected continuously for several weeks from rabbits by radio-frequency telemetry. The experiments use the doxorubicin rabbit model, which has been previously used to model human heart failure [6]. The objective of the experiments at West Virginia University is to inject the rabbits with doxorubicin that will potentially lead to heart disease and eventually cardiac arrest. Therefore, if a rabbit suffers from cardiac arrest, large amounts of ECG data from weeks of monitoring will be available for study. This data can be processed in order to find the types of electrical events that occur as the condition of the heart deteriorates. The results on one experiment supplied the data used in this thesis.

Previously, the duration of ECG collection was restricted to several days due to storage limitations. Several studies have been done where continuous data was collected from animals for 24 hours. These studies have focused on effects of stimuli or study of a specific condition. For example, one particular study monitored high frequency changes in the ECGs of dogs for 24 hours [7]. The focus of that study was to investigate a condition called myocardial ischemia and recovery. Another study recorded continuous

ECG data from rats for two days after a surgical procedure [8]. One study, which collected continuous ECG data from monkeys, lasted 12 days [9]. The experiment described in the previous paragraph collects data for weeks, which is a much greater period of collection than previous studies. In addition, the experiment at West Virginia University, unlike previous studies, focuses on monitoring long-term changes as heart condition deteriorates with disease.

ECG data is commonly collected from human subjects by Holter monitoring. This type of monitoring requires the patient to wear a small tape recorder for 24 hours. People do not have to be in a clinical setting for this type of monitoring. They are equipped with the recorder which they later return for analysis [10]. The duration of these tests rarely exceeds 24 hours.

Different methods have been employed to process data gathered from Holter monitoring. Several years ago, a unique approach was taken to compensate for the lack of processor speed. At the time, the available analyzing equipment relied on a person to monitor the data and detect changes, which was quite unreliable. A hardware Holter-tape analyzer was designed to compare heartbeat templates. Data was passed through circuits to calculate correlation coefficients for template formation. The correlation equation was called the Pearson product-moment correlation coefficient, which is identical to the equation exhibited above for CWA [11]. The processing that took place with hardware in that experiment closely resembles the software implementation that is the foundation for this thesis. That software is described in the next section.

2.4 Thesis Foundation

Recently, students working at the West Virginia University Biomedical Signal Analysis Laboratory developed ECG processing software. The software's purpose was to scan ECGs, determine the common ECG waveform morphology, and generate templates exhibiting the occurrences of heartbeats that do not correspond to what is defined as normal for each animal.

In order to process ECG data, the data samples have to be extracted from the data files and individual heartbeats have to be located. The proprietary data files generated by the monitoring system do not merely contain the points collected. A program had to be written to translate the data files in order to gather the data points. The data points are then stored alone in a binary file. As the data points are extracted, initial signal processing is performed on the data in preparation for template formation.

The pre-processing that occurred as the samples are extracted has a few goals. The first goal is filter any noise from the signal. The noise encountered in the ECG originates from muscle activity and baseline fluctuations [12]. A favorable range for allowable frequencies is commonly known to be between $20Hz - 60 Hz$. The filtering is implemented by a digital bandpass filter, which is derived using the discrete time transformation of the analog filter prototype shown in Equation 5 [13].

$$H(s) = \frac{1}{(1 + s/\omega_L)(1 + \omega_H/s)}$$

Equation 5: Transformation for Analog Bandpass Filter

Taking the discrete time (z) transform generates the constants shown in Equation 6.

$$\begin{aligned}
 a_o &= \frac{\omega_L}{(\omega_L + 1)(\omega_H + 1)} & \omega_L &= \tan \pi f_L / f_s \\
 a_1 &= \frac{\omega_L - 1}{\omega_L + 1} + \frac{\omega_H - 1}{\omega_H + 1} & \omega_H &= \tan \pi f_H / f_s \\
 a_2 &= \left(\frac{\omega_L - 1}{\omega_L + 1} \right) \left(\frac{\omega_H - 1}{\omega_H + 1} \right)
 \end{aligned}$$

Equation 6: Calculation of Constants for Software Bandpass Filter

In these computations, fL , the lower frequency limit, is 20 Hz and fH , the upper frequency limit, is 60 Hz . The sampling frequency, f_s , is 1000 samples/sec . The resulting difference equation for the filter is given in Equation 7.

$$y_i = a_o (x_i - x_{i-2}) - a_1 y_{i-1} - a_2 y_{i-2}$$

Equation 7: Difference Equation for Software Bandpass Filter

In the difference equation, y_i is a sample of the filtered signal and x_i is a sample of the unfiltered signal [13]. This difference equation is implemented and the absolute value of each sample was taken to obtain the rectified filtered signal, $|y|$. The effects of the filtering can be seen in Figure 1, which illustrates a signal before it is filtered and rectified. Figure 2 shows the same signal after filtering.

Unfiltered ECG

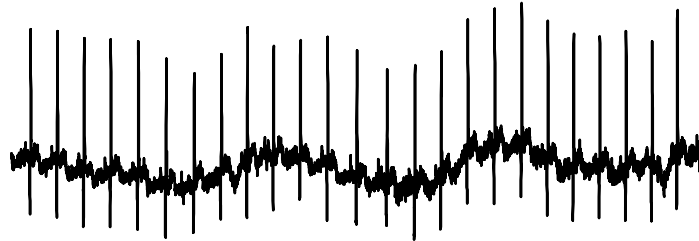


Figure 1: ECG with baseline drifting before bandpass filter is applied.

Filtered ECG

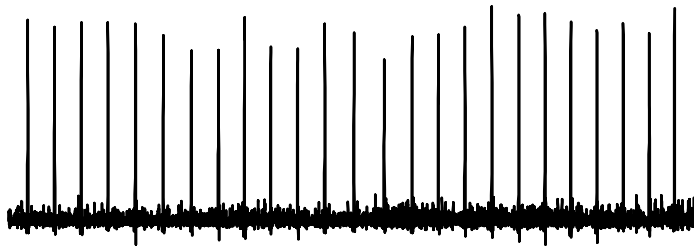


Figure 2: ECG after bandpass filter is applied.

The second objective of the pre-processing is to identify the location of individual heartbeats. This is achieved through a process called triggering [13]. Triggering records the location of one point for each heartbeat found in the data. These trigger points are then used to retrieve heartbeats for processing.

Triggering begins by establishing a threshold value. When a sample exceeds that value, it is defined as a trigger point for a heartbeat. In this case, the threshold is not

constant. The threshold will automatically adjust to account for beats of different amplitudes and shapes [13]. The threshold is updated by two mechanisms. Whenever the threshold is less than a fraction, b , of the signal, $|y|$, the threshold is reset to the value, $b|y|$. The second adjustment mechanism is a decay of the threshold in between the discovery of heartbeats. The decay is achieved by reducing the threshold level, v , with the comparison of each sample. The threshold is reduced to a level, cv , where the constant, c , is determined from Equation 8.

$$c = 2^{-1/tdfs}$$

Equation 8: Computation of Decay Constant for Software Trigger

The sampling rate, fs , is 1000 samples/sec , and td is the decay half-life. The threshold update function is now defined in Equation 9.

$$v_i = \max(b|y_i|, cv_{i-1})$$

Equation 9: Computation of Threshold Value for Software Trigger

The decay half-life, td , and the constant, b , have been known to be most effective when given the values 1 and 0.5 , respectively. However, these values can be altered to best fit an individual animal.

Once a trigger point has been found, a blanking period is enforced to ensure that more than one trigger point is not assigned to one heartbeat. The blanking interval is equal to the approximate number of samples for one heartbeat. After the blanking period passes, trigger point location resumes.

The trigger points are used to extract one heartbeat at a time. A heartbeat window is formed by a predetermined number of samples previous to the trigger point and a total size for the window. The desired window size was defined to be *150* samples with *50* of the samples located prior to the trigger point. The templates formed also consisted of *150* samples. This size could be varied from animal to animal.

The final objective of preprocessing is to generate a master template for each animal. This template is considered to be the normal ECG waveform for the animal. Averaging the first twenty heartbeats recorded for the animal forms the master template. Each heartbeat used to generate the template is compared to the averaged template. If one of the beats does not closely resemble the averaged template, it is discarded from the calculation and the average is recomputed. This process continued until no more beats can be discarded from the average.

The master and abnormal templates are formed by averaging heartbeat windows. When a heartbeat window is averaged with a template, each sample of the window is averaged with the corresponding sample of the template. The first sample of the window is averaged with the first sample of the template, the second sample of the window is averaged with the second sample of the template, and so on. When a template is first created, the samples that are retrieved around one trigger point form the template. For example, when creating the master template, the samples of the first heartbeat window for the animal are stored as the master template. When the second heartbeat is found, the first sample of the second heartbeat and the first sample of the template are added together. Then, the sum of the two samples is divided by two. This continues for every sample in the window. When the third heartbeat is retrieved, its samples are not simply

added to the averaged sample of the template and divided by two. Every sample of the template is first multiplied by two since the template has thus far been formed by two heartbeats. Then, each sample of the third heartbeat window is added to the corresponding sum for the template and the total is divided by three. Furthermore, when the fourth heartbeat is retrieved, each averaged template sample is first multiplied by three. Then, each sample of the new heartbeat is added to the matching sample of the template and divided by four. This process continues until twenty heartbeats have been averaged into the template.

Each heartbeat window is initially correlated with the master template formed in pre-processing. The correlation coefficient is calculated with CWA for each heartbeat multiple times because the heartbeat window gathered around a trigger point might not correctly align with the heartbeat of the template. The window has to be shifted to find the best alignment with the template. The shift value was chosen to be 15 samples. Therefore, each heartbeat window is shifted 15 samples to the left and 15 samples to the right to find the alignment that results in the highest correlation with the master template. If the heartbeat closely matches the master, the master's count is incremented. A heartbeat is considered to match the master template if the resulting correlation coefficient is greater than a predefined correlation threshold. Ideally, this threshold is between 0.8 and 0.9.

If the heartbeat does not match the master closely, the heartbeat is identified as an abnormality. If previous abnormalities have been stored for the animal, the same correlation process described for the master template is performed with each template until every template is checked or until no more templates exist. If the beat closely

matches one of the previous abnormalities, the trigger point of the shifted window that provides the highest correlation and file number in which the heartbeat occurs are recorded. The template's count is incremented and, if the template occurred less than 50 times, the samples of the new window were averaged with the samples of the template.

If no previous abnormalities are found for the animal or the window does not highly correlate with any existing template, the heartbeat is recorded as a new template. The information including its samples, a new label, the file in which the heartbeat is found, and the trigger point at which it correlates the highest with the master template are recorded. In addition, if an abnormality occurs at 50 consecutive trigger points, that template is reassigned as being the new master template. A basic flowchart for the correlation for each heartbeat is shown in Figure 3.

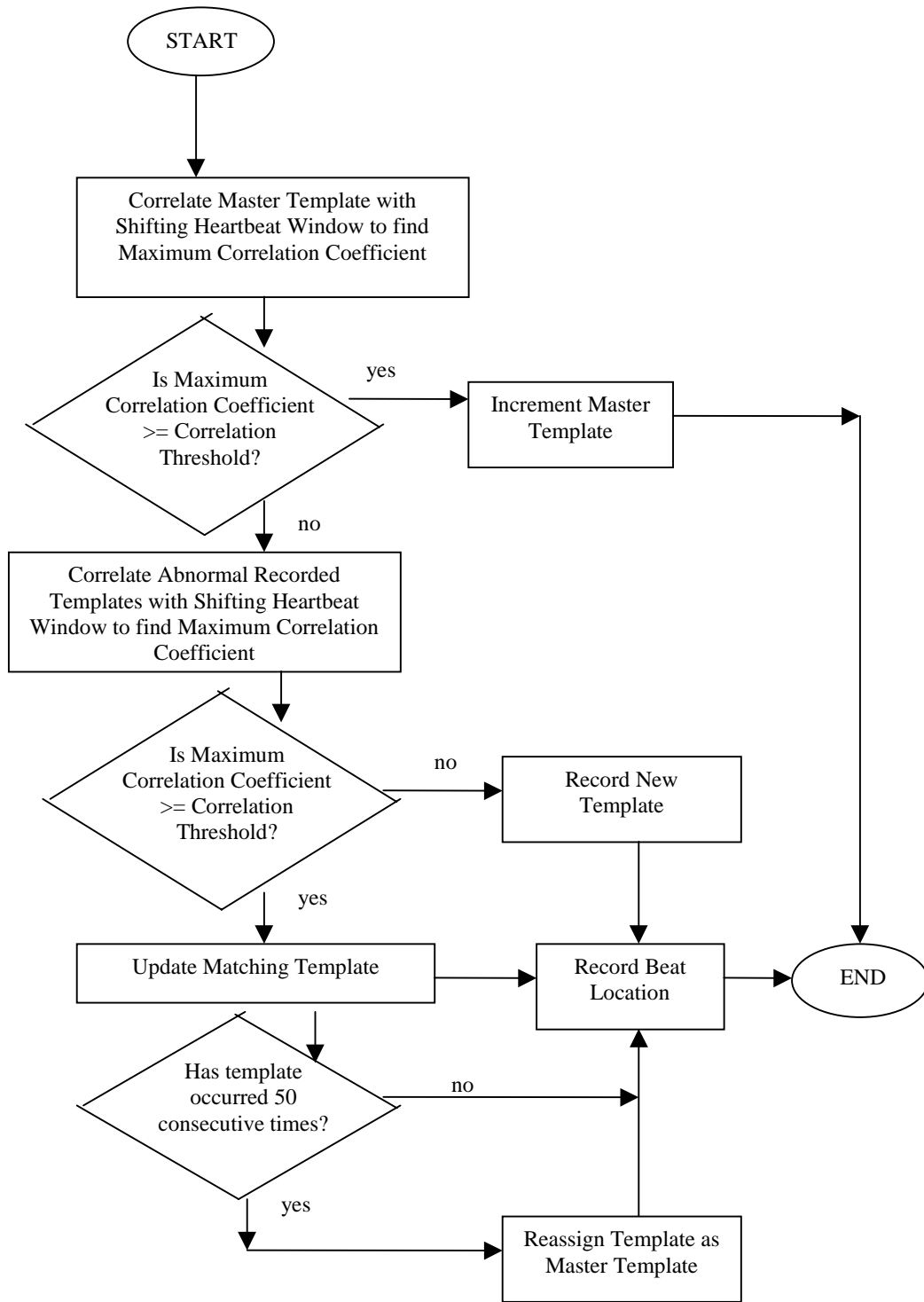


Figure 3: General Overview of ECG Template Generation

All files generated are binary files. Additional programs were written to translate the binary files into text files that can be studied by researchers. The code was previously written in C on an MS Windows platform. This software served as the starting point for this thesis research. To begin, the application was ported to Linux to prepare for distributed processing on the Beowulf cluster.

2.5 Parallel Algorithms

Parallel algorithms can be categorized as being synchronous or asynchronous, coarse or fine grained, and bandwidth greedy or frugal. Asynchronous algorithms are usually easier to implement than synchronous ones. Synchronous algorithms require strict organization to keep multiple processes synchronized. Grain size refers to the amount of work performed by individual processes, which relates to the size and number of data blocks given to each process through communication. Small grain size is characterized by small data sets being communicated frequently. Large grain size means larger data sets are communicated less frequently. Larger grain sizes are preferred because they reduce the ratio of communication to computation. Bandwidth frugal algorithms are also preferred. A lot of network communication can greatly reduce the speed of an algorithm. Obviously, a favorable algorithm is one that limits its communication over a network. Furthermore, parallel algorithms can be classified as having regular or irregular data structures and distributed or shared memory. Regular data structures are those that can be easily divided among processes. Data structures such as trees or linked lists are considered to be irregular data structures. Lastly, address space

is a very important factor in parallel algorithms. Shared memory configurations reduce the complexity of parallel algorithms. Beowulf clusters have distributed memory. Information is shared between processes through message passing over a network [14].

The ideal parallel application is one that has process-level parallelism. Process-level parallelism means that an identical sequential program is executed independently on multiple processes. The results of one process do not affect the results of any other process. These applications are referred to as being "embarrassingly parallel" because no changes need to be made to existing sequential code [14].

2.6 Message Passing Libraries

In order to parallelize the ECG template generation software for this research, there has to be a way for processes running on separate machines to communicate with each other. If different processes are analyzing separate ECG data sets, the processes need to share the templates that are being generated for an animal. Also, a process often needs to know when other processes have completed processing. Message-passing libraries make the communication between processes on different machines possible. They provide functionality to easily pass messages between processes.

Before the 1990s, many parallel programming libraries existed. Some were developed by commercial manufacturers specifically for their own massively parallel computer systems. Others were designed for specific research projects. Most of these libraries were very similar in functionality, but because of different syntactical styles, it was very difficult to write applications that were portable across the platforms. In 1992, the Center for Research on Parallel Computing at Rice University sponsored a workshop

on the Standards for Message Passing in a Distributed Memory Environment. The workshop spawned a two-year series of meetings and discussions that led to the development of the MPI-1.0 Message Passing Interface. The objective of the specification was to improve efficiency, portability, and functionality in parallel programming. The developers of the specification included all the best features from existing message-passing libraries [14].

MPI's basic concepts include initialization, simple point-to-point communications, message buffers, message families, process naming, communicators, collective communications and many more. An MPI application has to be initialized and finalized. The first three function calls in an MPI program are:

```
MPI_Init(&argc, &argv);  
  
MPI_Comm_size(MPI_COMM_WORLD, &size);  
  
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

The `MPI_Init()` call is passed command line arguments that are supplied by the user. These arguments specify the number of processes as well as the names of the hosts on which to run. The size variable defined by `MPI_Comm_size()` lets the process know how many processes are running. `MPI_Comm_rank()` defines the rank of the process, which is simply a unique number given to the process to distinguish it from the other processes. The size and rank are two very useful parameters for every process. MPI applications are typically generated with a single copy of source code. Usually, a process performs operations based on its rank. The `MPI_COMM_WORLD` variable is the process's communicator. The communicator is the mechanism that allows a process to communicate with other processes in the same group. All MPI processes start out in the

MPI_COMM_WORLD group, but they may be gathered into different groups using different communicators [15].

The basic send and receive operations of MPI resemble the equivalent operations from other libraries and socket communication. The MPI_Send() and MPI_Recv() functions require parameters for the data buffer, buffer size, destination, and tag as well as the communicator and MPI data type for the data being sent. These functions can be applied in a variety of ways. They can be applied in blocking mode, which means that processing will not continue until the data is received or sent. They can also be applied in the opposite manner. One function, MPI_Sendrecv() can both receive and send a message with a single call. One of the most difficult lessons to learn when writing parallel programs with MPI is the concept of correctly ordering corresponding sends and receives to avoid deadlock.

One aspect of MPI and other message passing libraries that is very useful is the collective communications that include both data movement and computations across processes. Examples of the data movement operations are MPI_Scatter() and MPI_Gather(). MPI_Scatter() allows the individual elements of an array on one process to be distributed among the other processes. The MPI_Gather() operation does the opposite. It combines single elements from other processes into an array in the calling process. One of the most often used data movement operations is MPI_Bcast() which can take a value from one process and "broadcast" it to the other processes in the group.

An example of a collective computational operation is MPI_Allreduce(). The function is called by

```
MPI_Allreduce(void*send, void*recv, int count, MPI_Datatype type, MPI_Op
op, MPI_Comm comm)
```

where *send* points to the data being sent, *recv* points to the buffer waiting for data, *count* is the number of items being transferred, and *type* is the data type. The *MPI_Op* parameter gives this operation diverse functionality. *MPI_Op* can have values such as *MPI_SUM*, *MPI_MAX*, *MPI_MIN* and others. If *MPI_SUM* is used, the *send* data from all processes is added and the sum is returned to the *recv* buffer of all calling processes [15].

The functions described here only begin to show the functionality provided by MPI. The library possesses 125 functions that provide the means for building a wide range of parallel applications. A favorable trait of MPI, however, is that knowing just a few functions like those described here allows for a person to write successful parallel algorithms.

Another well-known message passing library is Parallel Virtual Machine (PVM). Two popular versions of PVM were developed in the past. The first was PVM 2.4.x [16]. In most aspects PVM 2.4.x, is very similar to MPI. It has a corresponding function to most of the basic MPI calls such as the basic send, receive, initialize, and finalize functions. One difference is that the MPI initialization and finalizing calls are always required, which is not the case with PVM 2.4.x because its applications are invoked in a different manner. Another difference is that PVM 2.4.x doesn't have a non-blocking receive or possess a set of collective operations as extensive as MPI. PVM 2.4.x only has a couple functions that may be classified as collective operations [15].

The second PVM version, PVM 3.2.x [17], was an improvement over the previous version. PVM 3.2.x offers support for parallel machines. The previous version could only run on heterogeneous clusters of workstations or on the front end of parallel machines. The newer version does have a blocking receive and can implement more collective operations with dynamic groups [15].

Several other message-passing libraries exist. Some offer unique options that are not available in MPI. For example, the p4 library was designed at Argonne National Laboratory for message-passing models as well as shared memory [15]. The support for shared memory is an advantage over MPI. However, in general, the MPI libraries cover the functionality offered by other message-passing libraries well.

Chapter 3

Parallel Algorithms for Template-Based ECG Processing

The goal of this research is to investigate various methods to parallelize the ECG template generation software discussed in Section 2.4. The software utilizes CWA for morphology analysis. CWA is not favorable for use in ICDs because of its high computational demands. It is, however, favorable for this research because it has been found to be highly accurate in morphology analysis in previous studies, and it has consequently been used to evaluate new algorithms as they are developed.

In general, one must consider performance issues such as computational intensity and file I/O when developing software. When developing parallel algorithms, one must also take into account communication between multiple processes and scalability issues. Parallel applications are designed based on their data structure, synchronicity, grain size, bandwidth greed, and address space. All of these characteristics affect an algorithm's potential for parallelization.

The ECG template generation algorithm can be separated into two stages. The first stage is the pre-processing portion of the software that performs the data extraction, bandpass filtering, and triggering of the ECG. The pre-processing stage alone can be classified as having process-level parallelism. Separate processors can perform the stage's sequential algorithm for filtering and triggering concurrently and independently. The results from the pre-processing of each file do not affect the processing of other files on different processors.

The focus of research for this project is the second stage of the ECG software, which is the template generation algorithm using CWA. This stage lacks process-level parallelism. A single set of templates needs to be formed. The computation occurring on one processor is not independent of the results generated by other processors. Each process needs to not only access its own set of ECG data, but it needs to share a set of templates that are asynchronously generated by all processes.

The template generation stage is measured against the characteristics of parallel algorithms defined in Section 2.5. In all of the approaches presented in this document, the synchronicity, data structure, and memory model characteristics are the same. The algorithms are asynchronous. Each process can work on its own set of data at a different rate from other processes. The data is classified as regular data because the stream of ECG samples and separate templates could easily be divided into separate blocks for different processes. The algorithms have to work with distributed memory space since they are being distributed over a cluster of machines.

The other qualities such as bandwidth demand and grain size are varied in the different implementations of the template generation algorithm. The methods presented differ in their amount of communication and the way in which the processing load is distributed. The different approaches to parallelization of the application are described in detail in Section 3.1.

All of the parallel approaches have a couple more aspects in common. In each algorithm, each process is identified as a root or non-root process. Only one root process exists and it is typically responsible for coordinating other processes if needed. The root process is the process residing on the main node of the cluster. This main node is the

machine on which the program is being invoked and where all the files are stored. When the application is invoked on the root node, the processes on the non-root nodes are initiated.

All ECG and template files are stored on the root node. The files are shared to all machines via the Network File System (NFS) protocol. All non-root processes can access files as if they were local files, but the file input/output is actually occurring through the network. The advantage of using NFS is that it allows all machines to share a single file set. The algorithms are designed in such a way that no two processes will attempt to alter the same file. The disadvantage of NFS is that processing time will be lengthened by the network communication caused by file input/output.

3.1 Root-Only Template Formation (ROTF) Algorithm

The first approach in distributing the algorithm is designed with a goal of avoiding interprocess communication. Each process scans its own set of ECG files to find abnormalities. The root process, however, is the only process to form templates.

Every non-root process correlates all the trigger points in its assigned files with the original master template of the animal. If a heartbeat is found that does not match the master template, the file number and trigger point of the heartbeat are recorded in a file. When a process reaches the end of its file supply, it waits for all other processes to run out of files. After all processes, including the root, have completed processing their files, the non-root processes become idle.

The root process differs from the other processes in that it forms templates as it discovers abnormalities. After all processes exhaust their file supply, the root reads the

files containing the abnormality locations recorded by the other processes. Then, it retrieves the heartbeat windows corresponding to those locations and correlates the windows with the set of templates it created, including the master template. The master template is compared to the abnormalities again in case the root process had reassigned the master template during its template formation.

This implementation involves almost no network communication. The only communication between processes occurs at the end. Processes wait for each other to send completion signals.

A basic block diagram exhibiting the file I/O and interprocess communication of this algorithm is found in Figure 4. The solid arrows represent file I/O and the dotted arrows show the messages that are passed between processes. The only signal sent between processes is the completion signal, which is communicated by every process to every other process.

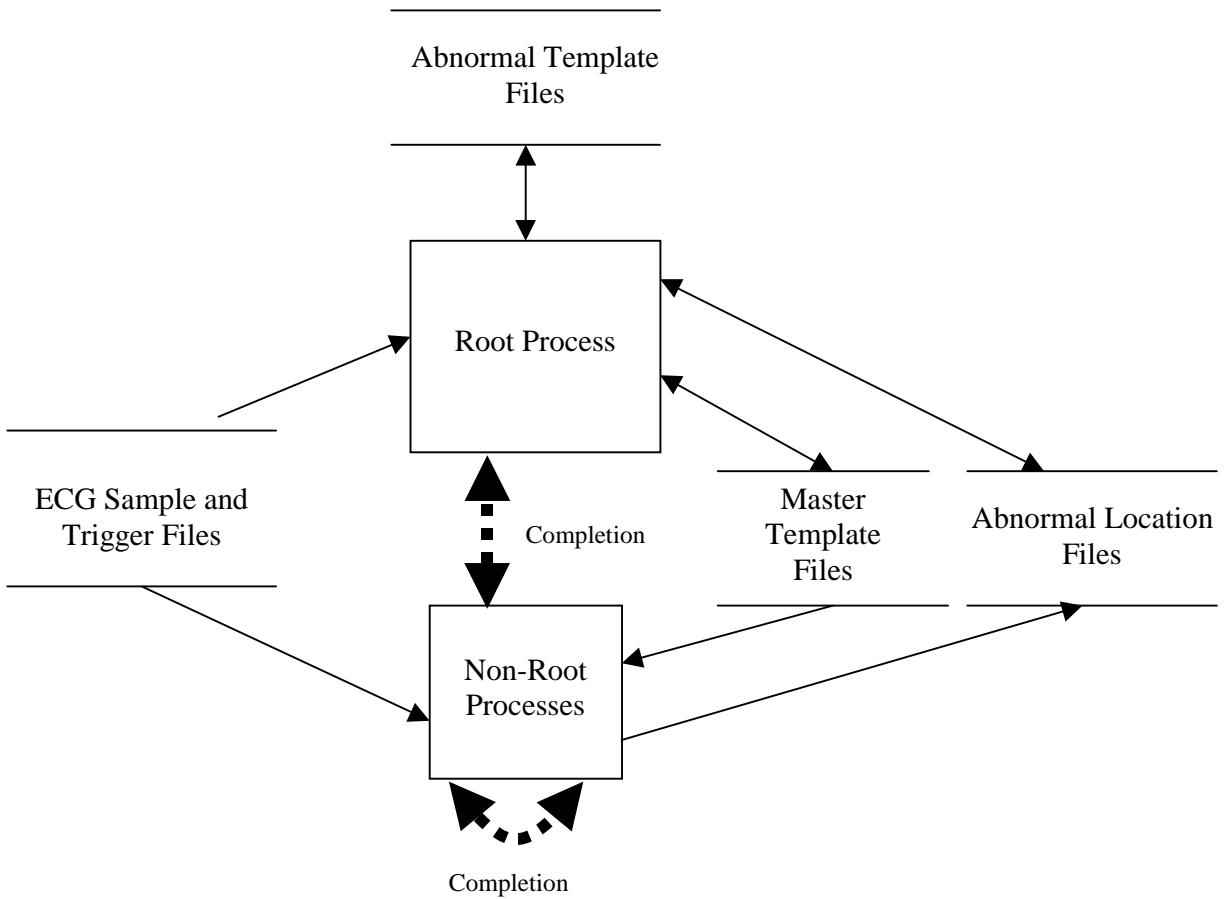


Figure 4: Block Diagram of Root-Only Template Formation (ROTF) Algorithm

A timeline is provided in Figure 5 to illustrate the points of synchronicity between processes. The dotted horizontal lines in Figure 5 represent periods of time when processes are not in synch with one another. The solid horizontal lines indicate the points at which the processes synchronize.

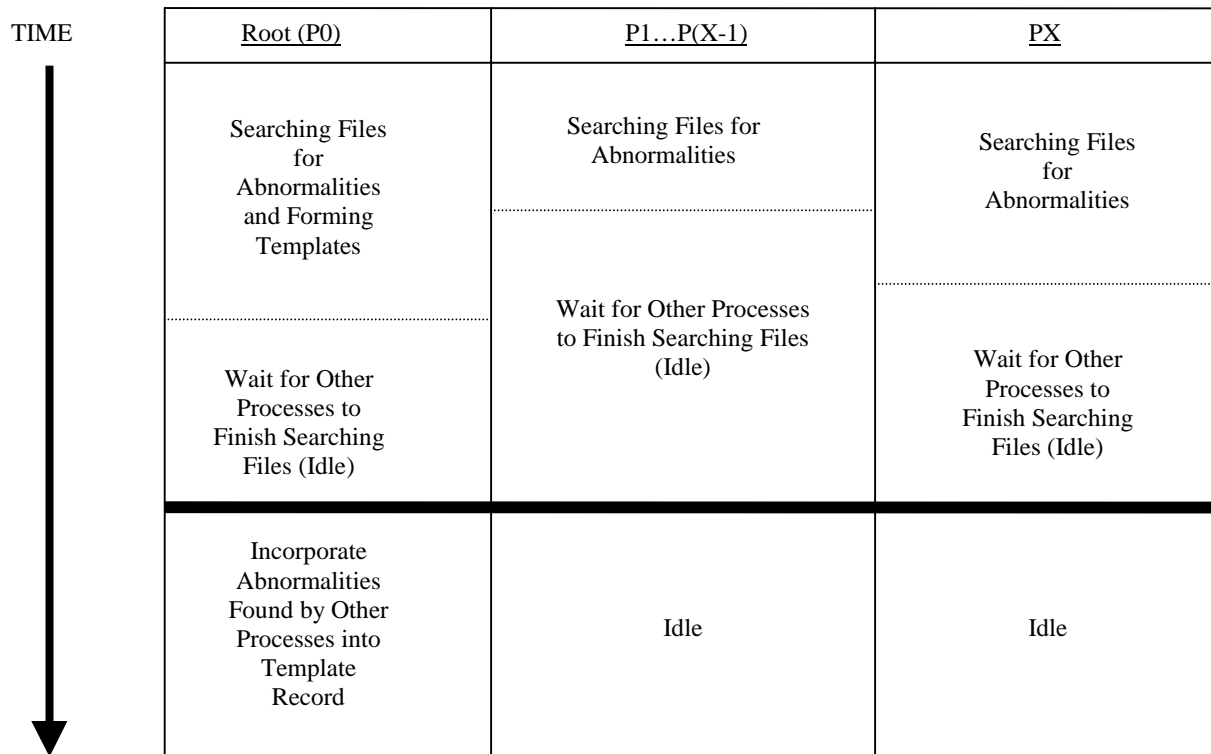


Figure 5: Timeline of Root-Only Template Formation (ROTF) Algorithm

3.2 Multiple Process Correlation Computation (MPCC) Algorithm

The second algorithm is aimed at distributing the calculation of the correlation coefficient. The single portion of the abnormality generation that is purely repetitive is the shifting of the heartbeat window to calculate the correlation coefficient multiple times. In this approach, the root processes all ECG files. Every time a trigger point is retrieved from a trigger file, the root process uses MPI_Bcast to issue the current sample file and trigger point to all processes. The root process also periodically sends a

completion flag to alert other processes when processing is complete. When the processes receive a true completion flag from the root process, they also end processing.

When a shifting correlation is performed, all processes, including the root, share the computation of the correlation coefficient. A window shifting value of *15* samples is used. This means that the correlation coefficient is computed *30* times because the window is incrementally shifted *15* samples forward and backward. Each process increments a counter from *-15* to *15*. The shift value is divided by the total of processes, and if the remainder is equivalent to the process's rank, that process computes the correlation coefficient for that window. After all the shifted coefficients have been calculated, the processes utilize the MPI function call, `MPI_Allreduce`, which will determine the overall maximum correlation for the trigger point and distribute it to all processes.

After the maximum correlation has been found, the corresponding shift point needs to be delivered to the other processes. Once a process receives the maximum correlation coefficient from `MPI_Allreduce`, it compares that coefficient value to the maximum coefficient it computed itself. Then, the process that had found the overall correlation coefficient sends the corresponding shift point to the other processes. Once this is complete, the root process can retrieve the heartbeat window from that shift point for forming templates.

This method contains a considerable amount of communication with small grain size. The root frequently sends small data sets to the non-root processes. This will result in a high communication to computation ratio, especially when compared to the ROTS algorithm.

A general block diagram of this implementation can be found in Figure 6. The solid lines represent the file I/O and the dotted lines illustrated the communications between processes.

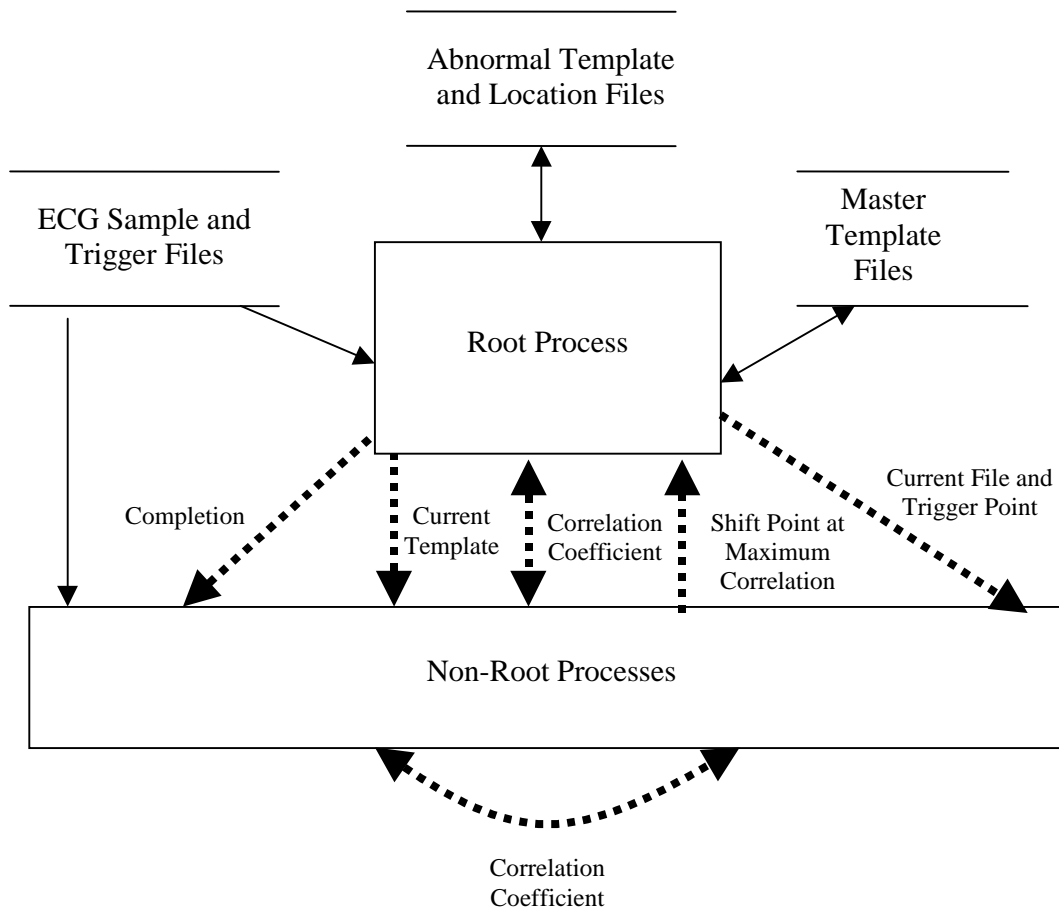


Figure 6: Block Diagram of Multiple Process Correlation Computation (MPCC) Algorithm

A timeline is provided in Figure 7 to illustrate the points of synchronicity between processes. The dotted horizontal lines represent periods of time when processes are not synchronized and the solid line indicate points of synchronization.

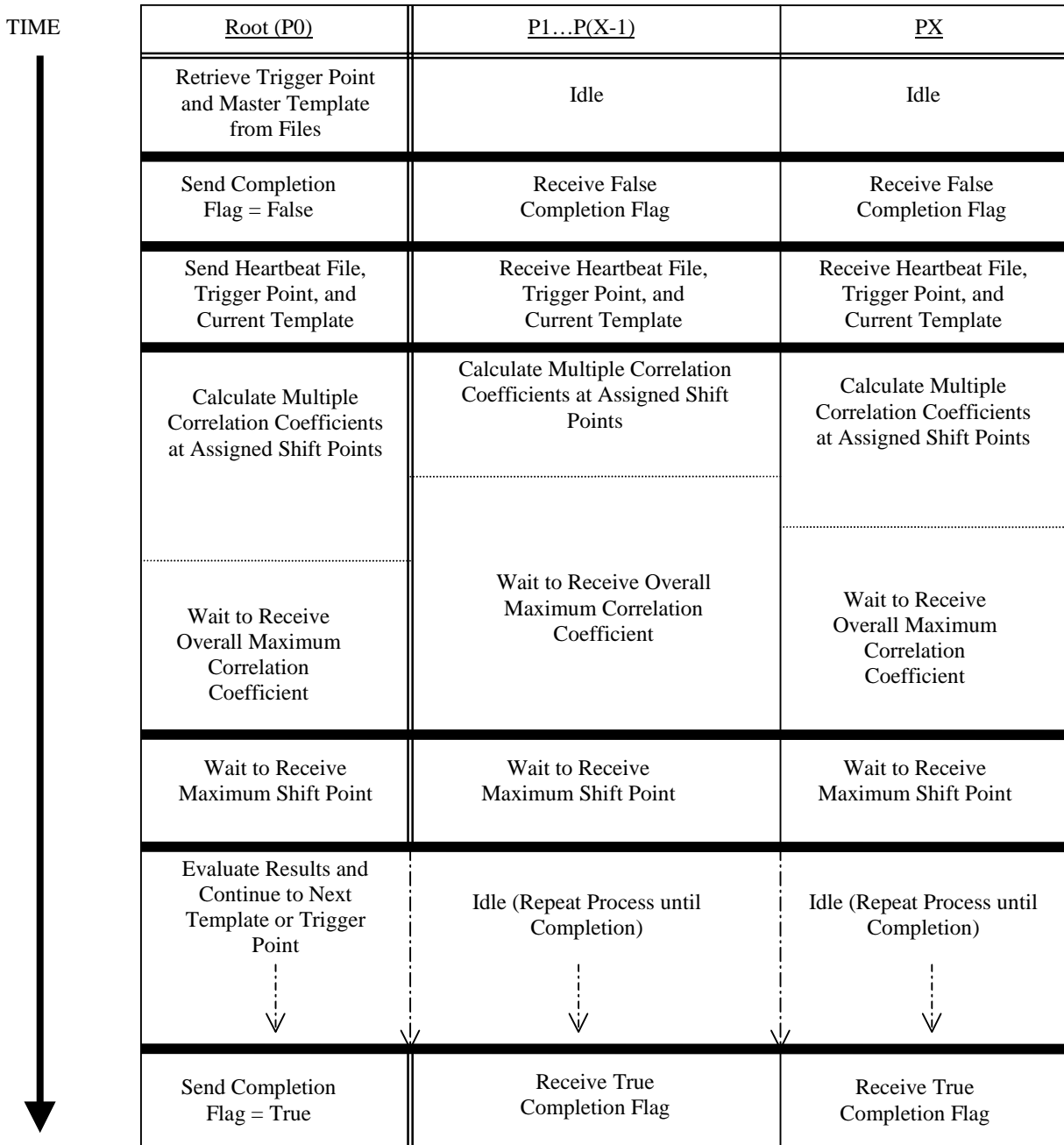


Figure 7: Timeline of Multiple Process Correlation Computation (MPCC) Algorithm

3.3 Node Data Server (NDS) Algorithm

In the third algorithm, the processing of the sample files is distributed among all processes except the root node. Each process scans its own set of samples and forms templates. However, the template set and record of abnormality locations are shared among the processes. The root acts as a data server for the other processes. The root controls all retrieval of templates, and any additions or modifications to the template record are implemented through the root. The root records the status of the processes, so that no overlap in the template generation or modification occurs. The communication between processes in this method was achieved with the strict ordering of the MPI communication functions, MPI_Send and MPI_Recv.

The root process waits to receive a message from any non-process. Another process will initiate communication with the data server by sending an instruction. A small set of instructions are defined to enable processes to receive templates from the data server and also send the server updates to the template record. This set of instructions is summarized in Table 1.

Instruction	Source Process Responsibilities	Root Process Responsibilities
MASTER_REQUEST	--receive flag to determine if master template has change --if master has changed, receive master template	--send flag to alert source if master template has changed --if master has changed, send master template
TEMPLATE_REQUEST	--receive next template for correlation	--if another template exists, send template --if another template does not exist, send false template with label of -1000
NEW_TEMPLATE	--receive flag to see if a new template has been created by another process since retrieving the last template --if flag = SEND_TEMPLATE, send samples and new template location of event to root (else, continue retrieving templates)	--send flag to alert source if a new template has been formed by another process since the last template was sent --if no new template was formed by another process, receive new template and location of event --record new template and location
INCREMENT_TEMPLATE	--send location of event	--increment template's count --receive location of event --record updated template and location
AVERAGE_TEMPLATE	--send averaged template samples --send location of event	--receive averaged samples --receive location of event --record updated template and location
MASTER_MODIFY		--increment master template's count
PROCESSING_COMPLETE		--increment number of completed processes --if number of completed processes = total number of processes, exit

Table 1: Instruction Definitions for Node Data Server (NDS) and Thread Data Server (TDS) Algorithms

A non-root process can request the master template from the data server by sending MASTER_REQUEST. When MASTER_REQUEST is sent, the data server sends a flag to the process to say that the master template has or has not been reassigned. If the master has been reassigned, the new master template is sent to the process. Otherwise, the non-root process continues with its previous copy of the master template.

When an abnormality is detected a non-root process, the process issues the `TEMPLATE REQUEST` command. As a result of this command, the data server will check its record of the calling process's status. The data server determines what template was last sent to the process. Then, if another template exists, the template is sent to the process. If no other template exists, the server will send an empty template with a false label that the process will recognize. In addition, when the server sends a template to a process, it checks the status of every other process. If another process is processing the same template, a flag is set to signify that the processes are sharing a template. Therefore, when one process tries to alter the shared template, the template will be updated for the sharing processes as well. This means that the averaged samples of the template and its count will stay current for all processes.

When processes are modifying templates, they can update the template record using `NEW_TEMPLATE`, `AVERAGE_TEMPLATE`, `INCREMENT_TEMPLATE`, and `MASTER MODIFY` instructions. The data server maintains a label for the most recent template created. When the server receives the `NEW_TEMPLATE` signal, the server checks the last template retrieved by the calling process. If that last template has the same label as the most recent template created, it receives the new template found by the process and adds it to the template record. If, however, the last template sent to the process is not the most recent one created, that means that another process has created a new template since the current process retrieved its last template. In this case, the server notifies the processes to keep retrieving templates.

When `AVERAGE_TEMPLATE` or `INCREMENT_TEMPLATE` is sent, the server performs the appropriate actions to either average or increment the template

altered by the calling process. If another process is also using the template, its copy of the template will be updated also to ensure that the data stays current. When MASTER_MODIFY is received, the server simply updates the count of the master template. In addition, if any template other than the master is found to have occurred 50 consecutive times from one process, the server reassigns that template as the master template. A flag is set for each process, so the next MASTER_REQUEST signal will return the new master to each process. When a process has exhausted its file supply, it issues the PROCESSING_COMPLETE instruction, and the server tracks the number of completed processes.

This implementation is similar to MPCC in that it relies heavily on communication and uses small grain size. However, NDS may serve to be a better approach than MPCC because it distributes the processing of files over more than one processor.

3.4 Thread Data Server (TDS) Algorithm

This algorithm is a modification of the NDS algorithm. The goal of this approach is to eliminate the uneven distribution of files to processors. This method allows the root node to act as a data server as well as allowing it to process its own set of samples. A separate thread is created on the root node of the cluster. This thread is utilized as the data server. Another process running on the root node behaves like all of the other nodes of the cluster.

In this case, the communication between the data server and the other processes is achieved through sockets. The communication in the NDS algorithm allows the root

process to receive a message from any source. All processes are always bound to the same communicator. With this socket implementation, the server waits to accept a new connection from any source after each instruction cycle. A client socket is reconnected to the server with every instruction issued. Everything else, including the instruction set given in Table 1, is the same as in the NDS algorithm.

The block diagram for the NDS and TDS algorithms is shown in Figure 8 illustrating communication between processes and file I/O. The instructions MASTER_REQUEST, TEMPLATE_REQUEST, and so on are represented by the *Instruction* variable passed from the non-root processes to the data server. The data that is transferred as a result of the instructions such as flags and templates are grouped into the *Instruction-Specific Data* variable.

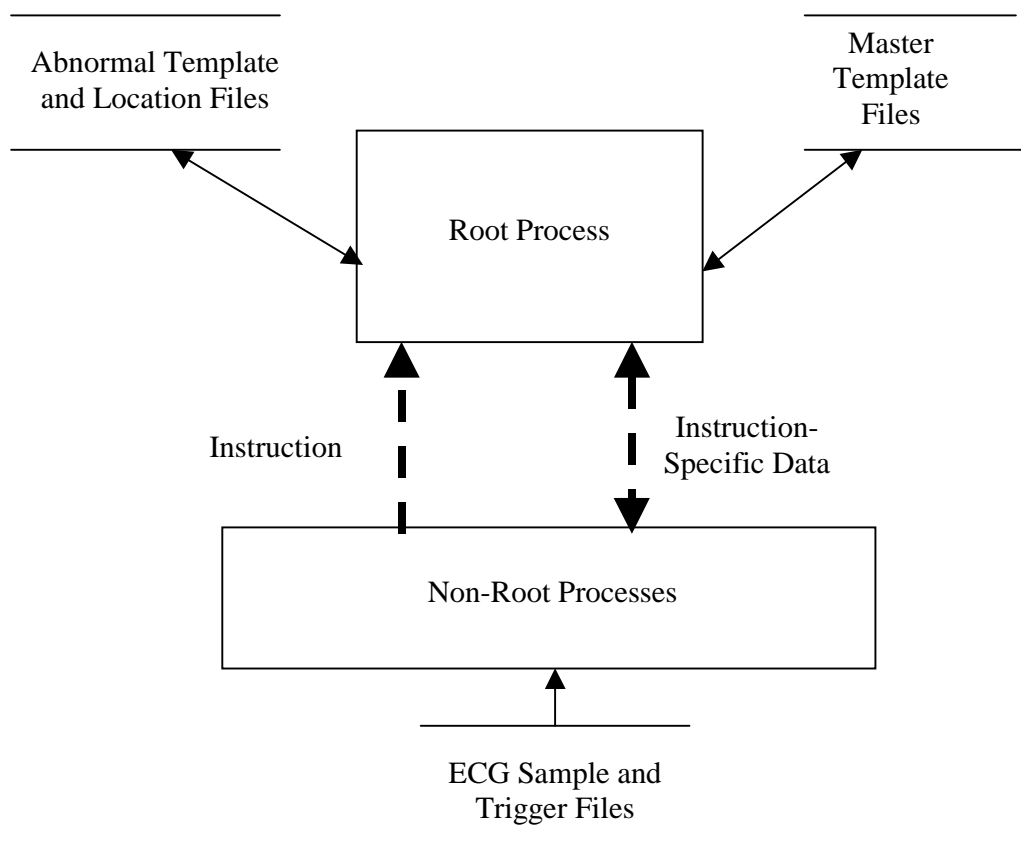


Figure 8: Block Diagram of Node Data Server (NDS) and Thread Data Server (TDS) Algorithms

A timeline is also supplied in Figure 9 to illustrate a possible scenario between processes. The dashed horizontal lines represent asynchronous computation. The solid lines represent synchronization for communication.

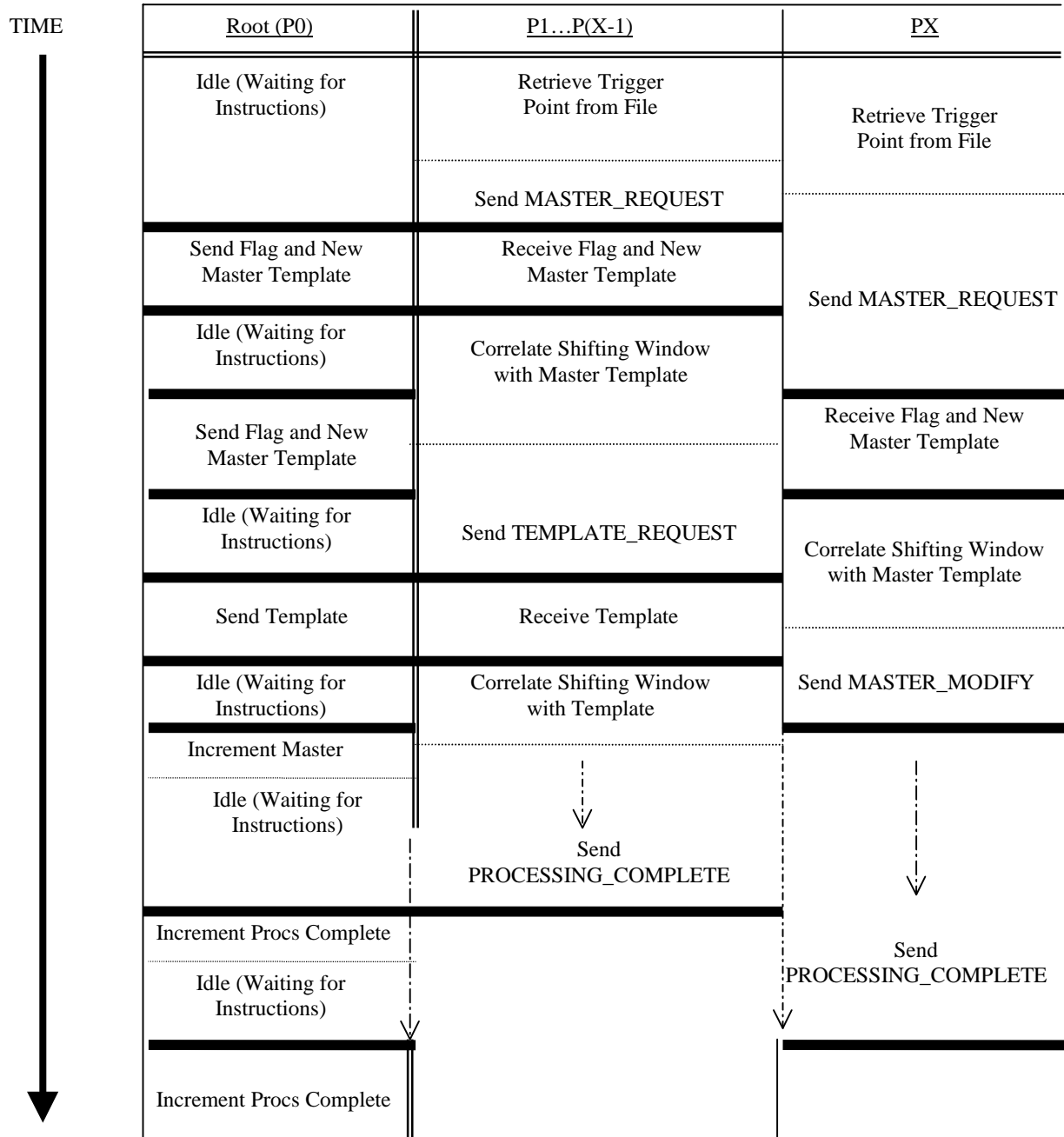


Figure 9: Timeline of Node Data Server (NDS) and Thread Data Server (TDS) Algorithms

Chapter 4

Results

This chapter contains the timing data collected during the testing of the parallel template generation algorithms and analysis of the results. The overall time for each method is presented as well as a description of how the time is broken down into different components. Measurements were taken to separate time spent in the communication, computation, file I/O, organization, and idle states. The meaning of the organization and idle states will be defined for each algorithm. Charts and graphs are available for each parallel algorithm to aid in the evaluation of the different approaches.

Section 4.1 provides timing data for each individual algorithm. The data given in the tables of the section resulted from processing six files. In Section 4.2, the results of the different approaches are compared with one another. Lastly, some templates that are formed by the algorithms are displayed in Section 4.3.

4.1 Time and Efficiency Analysis

The four parallel algorithms are tested with varying numbers of processors and files. Testing the algorithms with varying numbers of processors permits analysis of the scalability of the algorithms. Predictions can then be made about the impact of adding additional processors. By testing the algorithms with a varying number of files, the consistency of each algorithm's performance can be evaluated.

Since the current cluster consists of six computers, the data sets are processed in groups of six files. The best way to analyze the algorithms is to test them under well-

balanced workloads. Therefore, if six processes are utilized and each process scans one file at a time, a multiple of six files is needed for an equivalent workload. In two of the four algorithms presented, all six processors do process on file at a time. The remaining algorithms, MPCC and NDS, are not the same. Since one machine is reserved as a data server in NDS, only 5 processes are analyzing files. However, in order to compare the performance of NDS with the performance of the other algorithms, it must also be test with groups of six files. In MPCC, one processor analyzes all files. Therefore, MPCC could be easily tested with any number of files.

All files reside on the root node for each algorithm. In most cases, the root node is the only process that modifies files. One exception is found in the ROTF algorithm. In ROTF, every process records the location of abnormalities in a file. There exists a separate abnormality location file for each process, so one process cannot write to the same file that another process modifies. In every algorithm, all non-root processes retrieve waveforms by accessing the sample files and trigger files through NFS. The samples and trigger files are never modified. Every algorithm is designed to restrict the modification of the template files to the root process.

4.1.1 ROTF Results

In ROTF, every processor scans its own set of files for abnormalities. Every process stores the location of the events, but only the root process (P0) forms templates. The organization statistic, *Org Time*, given in Table 2 describes the percentage of the overall time that P0 spends combining the abnormalities found by other processes with the templates. The idle statistic shows the amount of time each process has to wait for

other processes to complete processing their own files. It also contains the time that the non-root processes (P1-P5) were idle while P0 combined abnormalities.

# of Processors	Process Rank	Overall Time (min)	Comm Time(%)	Comp Time(%)	File I/O Time(%)	Org Time(%)	Idle Time(%)
1	P0	5.475	0	69.59	28.00	0.01	0
2	P0	2.750	0	69.13	27.84	0.04	0.60
	P1		0	70.01	27.55	0	0.03
3	P0	1.866	0	69.49	28.00	0.1	0
	P1		0	68.05	26.77	0	2.84
	P2		0	68.19	26.82	0	2.65
4	P0	1.829	0	69.56	27.98	0.07	0
	P1		0	69.35	27.26	0	1
	P2		0	35.00	13.76	0	50.04
	P3		0	36.04	14.18	0	48.54
5	P0	1.827	0	69.53	27.95	0.12	0
	P1		0	34.79	13.70	0	50.32
	P2		0	35.05	13.78	0	49.96
	P3		0	36.09	14.19	0	48.48
	P4		0	34.75	13.66	0	50.40
6	P0	0.943	0	68.40	27.54	0.25	1.46
	P1		0	67.37	26.52	0	3.80
	P2		0	67.85	26.72	0	3.10
	P3		0	69.85	27.52	0	0.23
	P4		0	67.26	26.47	0	3.95
	P5		0	67.06	26.40	0	4.24

Table 2: Data Collected using the Root-Only Template Formation (ROTF) Algorithm (Results from Processing Six Files)

These results exhibit that this algorithm does not rely on communication at all. Most of the time in each process is spent in computation. One can see the changes in times and processing loads as the six files are distributed over an increasing number of processes. The most dramatic decreases in processing time occur when the second, the third, and the sixth processor are added. These sharp decreases occur for obvious reasons. When a second processor is added, *P0* is relieved of half of its workload. It processes 3 files instead of six. When a third process is included, the workloads of *P0* and *P1* are reduced by 30% from three files to two files. A subtle decrease is found when the fourth and fifth processes are added because there is still at least one processor that

has to process 2 files. When the last processor is added, the significant drop in time is attributed to an equal distribution of files over the processors.

The percentage of idle time for non-root processes drastically increased until the final processor was included. The idle time of most processes is significantly reduced when the workload is equally distributed. The organization percentage recorded for the root process steadily increased with the addition of more processes. The root has to access more abnormality location files as more processes are used.

The same trends are discovered when twelve or eighteen files are processed because the number of files is a multiple of the number of processes. The results of the experiments where the number of files was increased are provided in Figure 10.

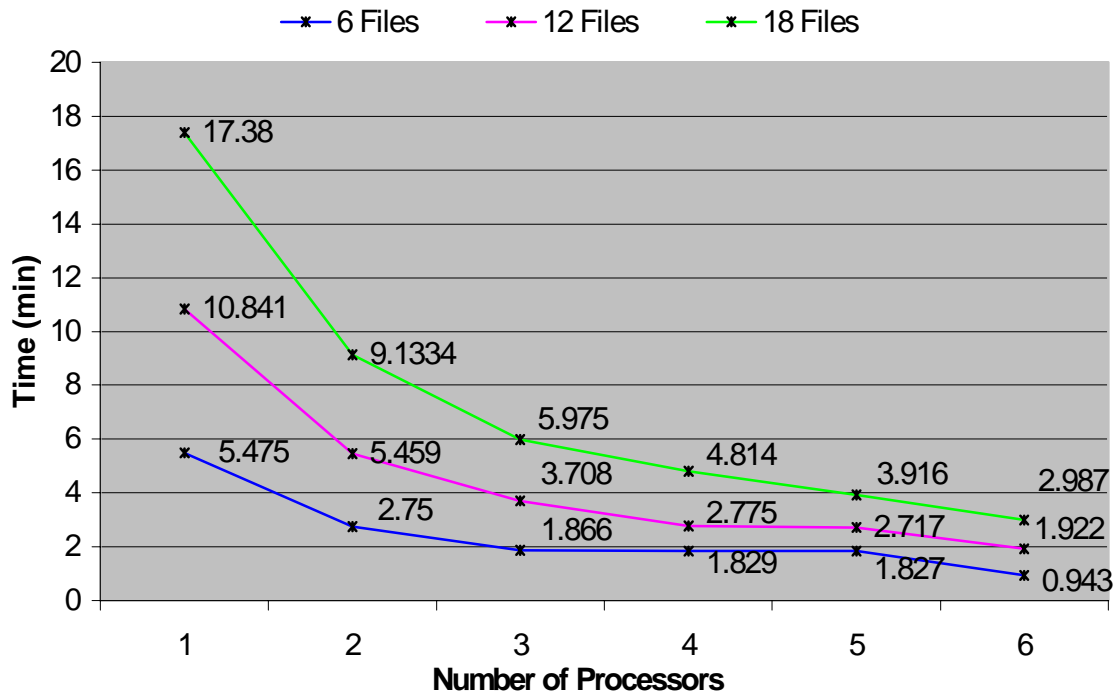


Figure 10: Results for Root-Only Template Formation (ROTF) Algorithm

4.1.2 MPCC Results

In the MPCC algorithm, the root process scans all of the files. The other processes share the calculation of the correlation coefficient. No organization component is present in the approach. The idle times are logged from the non-root processes. They are considered to be idle in between communications from the root process because they have no tasks until the root gives them a window to correlate.

# of Processors	Process Rank	Overall Time (min)	Comm Time(%)	Comp Time(%)	File I/O Time(%)	Org Time(%)	Idle Time(%)
1	0	11.424	0.24	66.58	30.41	0	0
2	0	7.687	23.21	49.99	24.16	0	0
	1		19.96	50.94	22.28	0	4.19
3	0	6.331	35.99	40.70	20.78	0	0
	1		32.71	41.53	18.47	0	5.03
	2		32.95	41.31	18.44	0	5.05
4	0	6.280	53.08	29.09	15.66	0	0
	1		44.01	33.65	15.23	0	5.10
	2		40.97	35.49	15.19	0	5.87
	3		50.14	29.37	13.41	0	5.22
5	0	6.153	58.13	25.53	14.28	0	0
	1		53.79	25.99	11.96	0	6.51
	2		53.36	25.88	11.96	0	6.60
	3		54.03	25.93	11.92	0	6.36
	4		53.98	25.81	11.92	0	6.54
6	0	6.042	63.60	21.74	12.75	0	0
	1		52.92	26.52	12.21	0	6.56
	2		50.59	28.19	12.35	0	6.68
	3		59.53	22.08	10.33	0	6.43
	4		64.80	17.93	8.58	0	6.77
	5		65.52	17.88	8.54	0	6.55

Table 3: Data Collected using the Multiple Process Correlation Computation (MPCC) Algorithm (Results from Processing Six Files)

The shift value used for the template generation is 15 samples, which means that one process working alone has to calculate a correlation coefficient 30 times for each comparison between a window and template. MPCC exhibits a drastic decrease in processing time when a second processor is added. When a second process is added, the

amount of correlation coefficients computed on each process is divided in half. Each process is responsible for calculating coefficients for 15 shifted window positions. When a third process is added, the reduction of computations for each process is still significant. After that point, the increase in the number of processors has less impact on the processing time. The results will continue to become level as more processes are utilized. Once again, the same trend is observed when the number of files is increased. This is illustrated in Figure 11.

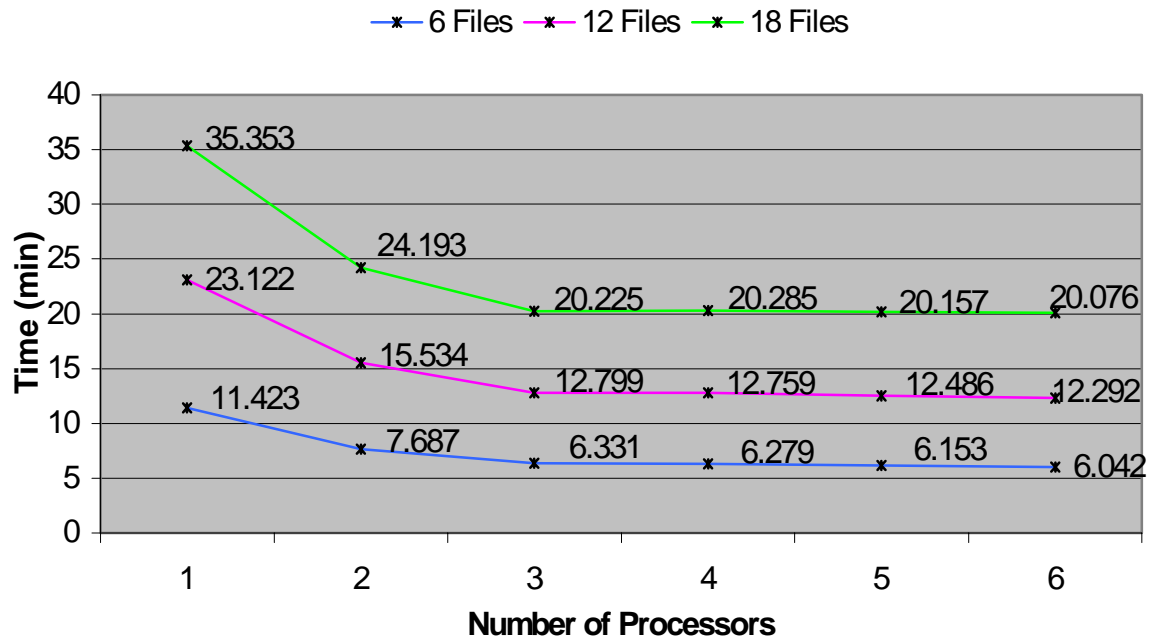


Figure 11: Results for Multiple Process Correlation Computation (MPCC) Algorithm

4.1.3 NDS Results

This NDS approach utilizes the root process as a data server for the remaining processes. Therefore data cannot be collected for 1 processor. There is also no organization component for this implementation. The root process, P_0 , spends the

majority of its time waiting to service requests from other processes. Therefore, the times recorded in Table 4 for the idle stage on the root process are high. The quick decrease exhibited with the ROTF method does not occur in this approach when the last process is added. That is because only 5 processors are processing the files, and there is at least one processor that still has to process 2 files. Otherwise, the performance improvements resemble those in ROTF.

# of Processors	Process Rank	Overall Time (min)	Comm Time(%)	Comp Time(%)	File I/O Time(%)	Org Time(%)	Idle Time(%)
2	0	6.279	1.74	0.19	0.00	0	97.45
	1		11.66	61.59	24.30	0	0
3	0	3.162	3.58	0.39	0.01	0	94.76
	1		11.96	61.19	24.14	0	0.24
	2		12.32	61.15	24.13	0	0
4	0	2.151	5.54	0.61	0.01	0	91.90
	1		12.17	61.22	24.17	0	0
	2		12.41	59.27	23.38	0	2.54
	3		12.89	59.35	23.41	0	2.02
5	0	2.114	5.66	0.61	0.01	0	91.77
	1		12.25	61.14	24.13	0	0
	2		12.62	60.18	23.73	0	1.16
	3		6.93	30.37	11.98	0	49.49
	4		7.56	31.28	12.36	0	47.55
6	0	2.108	5.78	0.63	0.01	0	91.57
	1		12.16	61.23	24.15	0	0
	2		6.53	30.26	11.94	0	50.05
	3		7.04	30.47	12.02	0	49.24
	4		7.63	31.37	12.39	0	47.35
	5		7.59	30.22	11.92	0	49.06

Table 4: Data Collected using the Node Data Server (NDS) Algorithm (Results from Processing Six Files)

As in the previous 2 cases, the same trend occurs whether 6, 12, or 18 files are processed.

These results are shown in Figure 12.

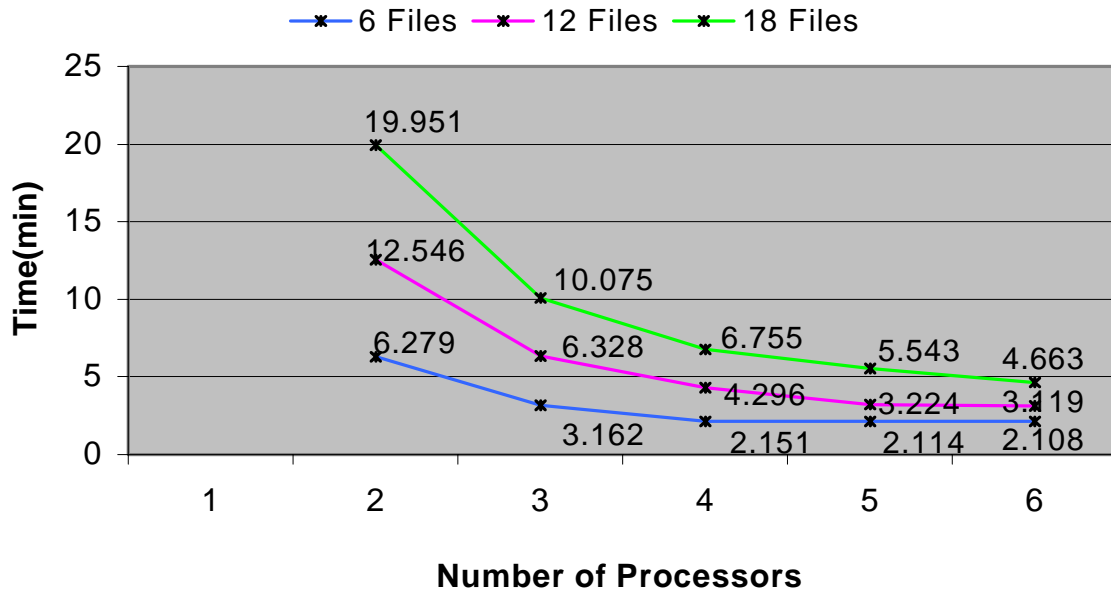


Figure 12: Results for Node Data Server (NDS) Algorithm

4.1.4 TDS Results

The TDS algorithm was created in an attempt at improving NDS. In NDS, the root process acted as a data server, leaving only five other processes to analyze files. In TDS, the root process creates a separate thread. This thread becomes the data server for the other processes. The original process running on the root machine behaves like the processes on the non-root machines. This method is implemented in hopes that the addition of the thread process will result in a shorter overall time than the NDS algorithm. Six files can be evenly distributed among six processes, one process on the root machine and one process on each non-root machines. Table 5 displays the timing results from TDS. In this case, there are seven processes instead of six. The data server process on the root machine is given the label *DS*.

# of Processors	Process Rank	Overall Time (min)	Comm Time(%)	Comp Time(%)	File I/O Time(%)	Org Time(%)	Idle Time(%)
1	0	6.138	16.49	55.75	25.3	0	0
	DS		5.55	0.21	0	0	91.47
2	0	14.51	3.64	12.06	5.50	0	78.26
	1		82.41	11.62	5.45	0	0
	DS		1.94	0.09	0	0	96.84
3	0	10.181	5.00	11.86	5.40	0	77.22
	1		83.48	10.92	5.12	0	0
	2		82.02	10.94	5.12	0	1.44
	DS		2.61	0.13	0	0	95.58
4	0	9.880	5.63	12.10	5.50	0	76.22
	1		83.01	11.24	5.26	0	0
	2		48.49	5.67	2.66	0	42.94
	3		45.76	5.84	2.74	0	45.41
	DS		2.69	0.14	0	0	95.41
5	0	5.826	9.75	21.03	9.63	0	58.64
	1		85.03	9.57	4.48	0	0.51
	2		82.16	9.62	4.51	0	3.29
	3		85.03	9.90	4.64	0	0
	4		84.38	9.55	4.47	0	1.19
	DS		4.65	0.25	0	0	92.01
6	0	5.612	7.74	10.96	5.03	0	75.78
	1		85.00	9.92	4.65	0	0
	2		77.20	9.99	4.68	0	7.69
	3		80.55	10.28	4.82	0	3.90
	4		81.96	9.91	4.64	0	3.06
	5		78.65	9.87	4.62	0	6.43
	DS		4.38	0.26	0	0	92.04

Table 5: Data Collected using the Thread Data Server (TDS) Algorithm
(Results from Processing Six Files)

The desired results were not achieved with this algorithm. Processing time actually increased dramatically when a second processor was added. The reason for this increase is based purely on the communication demand. Table 5 shows that the data server process still spends the majority of its time waiting for communication. The communication times for the non-data server processes are very high. The impact of increased communication far exceeds the benefits of adding the second processor. *P0*, which is a non-data server process, is different. Most of its time is in the idle stage. Its

communication demands are much lower because it resides on the same machine as the data server process and, therefore, its communication does not occur over the network. The communication performance is not favorable in this approach because of the repetitive reconnects that have to occur between the process and the data server for each instruction sent to the data server. In addition, the point-to-point communications that occur with sockets are not as efficient as those provided with MPI.

The trend that occurs as more files are added is displayed in Figure 13. As more processes are added to distribute the file load, the processing time begins to decrease slowly.

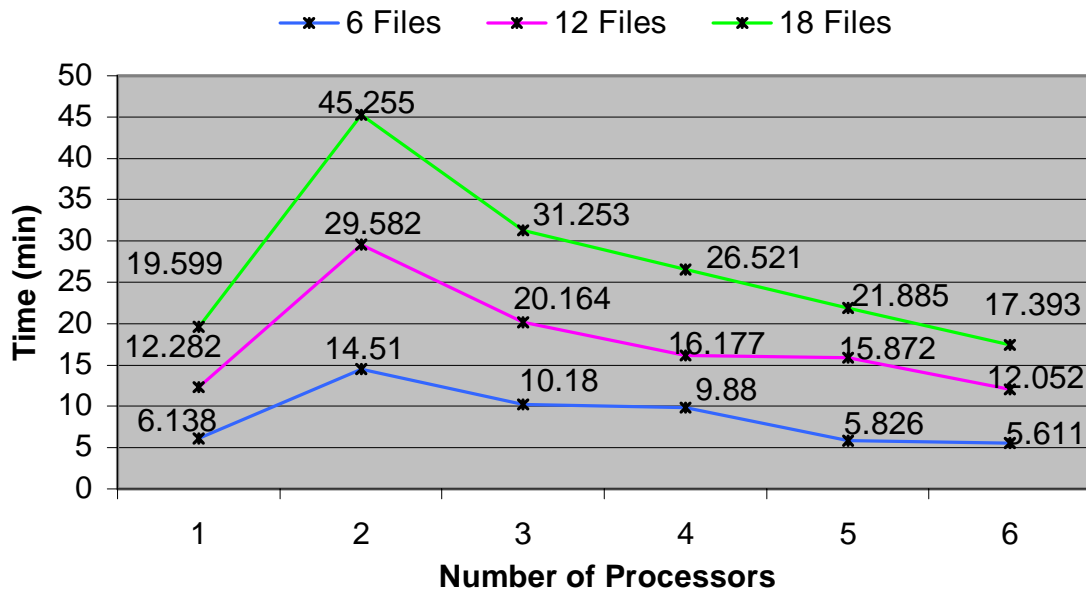


Figure 13: Results for Thread Data Server (TDS) Algorithm

4.2 Comparison of Methods

The graph provided in Figure 14 shows the results for all the algorithms presented. These times were collected during the processing of twelve files.

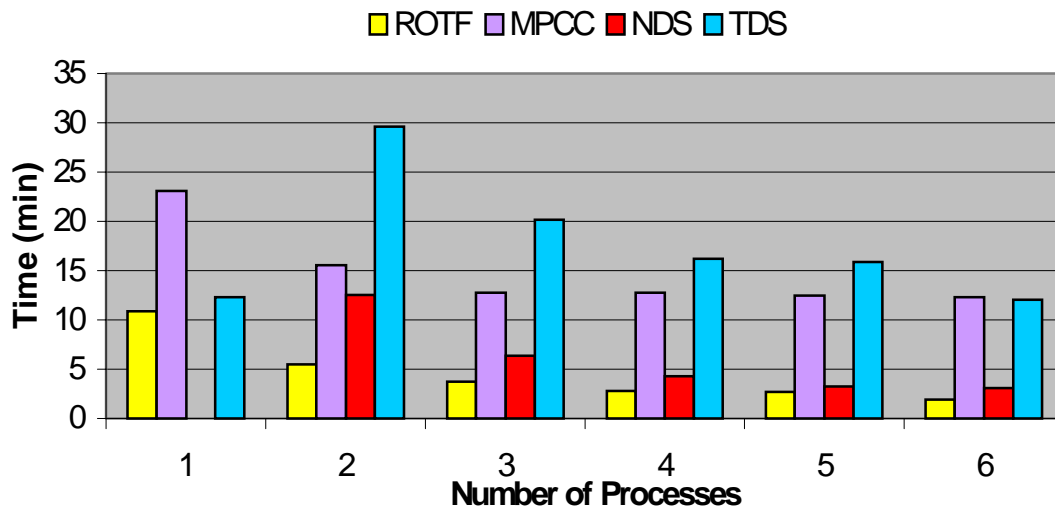


Figure 14: Comparison of Algorithms
(Results from Processing Twelve Files)

According to the results provided in Figure 14, the ROTF algorithm (shown in yellow) seems to have the best performance, exhibiting a speedup of approximately 5.5 between one and six processors. The next most favorable approach, according to Figure 14, is the NDS algorithm. It provided a speedup ratio over 4 just between two and six processors. Figure 15 shows only the results of ROTF and NDS. No result is given for NDS for one process in Figure 15 because the algorithm requires one machine to be a data server. Therefore, there has to be at least one other machine to process files.

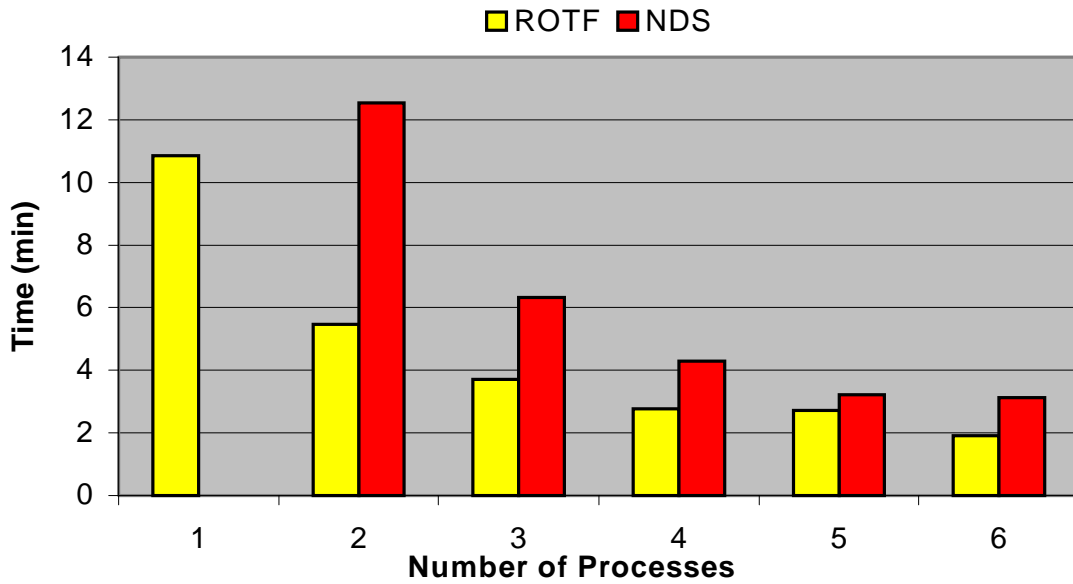


Figure 15: Comparison of ROTF and NDS (Results from Processing Twelve Files)

The results of NDS show that its performance increases at a higher rate than ROTF. The difference in times for the algorithms is quite large when two processors are utilized. However, the time difference becomes much smaller as more processors are included. The results for NDS seem to become level when five and six processes are used. As stated in Section 4.1.3, the uneven distribution of files to the processes in NDS is the cause of the similar results for five and six processors. This factor makes the comparison of NDS and ROTF difficult.

To further compare ROTF and NDS, refer to Tables 2 and 4. The time to process six files on six processes using ROTF is *0.943 minutes*. The time using NDS on six processes is *2.108 minutes*. There is a difference of *1.165 minutes* between the two algorithms. The NDS statistics supplied for six processes in Table 4 show that one process, *PI*, spends approximately twice the time of other non-root processes in the computation, communication, and file I/O stages. *PI* also has no idle time. This is the

result of the uneven file distribution. *PI* has to process one more file than the other processes. The other non-root processes are idle nearly fifty percent of the time. With these statistics, we can estimate the amount of time needed to process five files with NDS. Since no process would have to analyze two files, the time would be cut in half to approximately *1.054 minutes*. The time needed to process five files with ROTF would not differ from the time for six files. One process would simply be idle the entire time. Therefore, the ROTF time should remain approximately *0.943 minutes*. The time difference between the two algorithms now becomes only *0.1 minutes (6 seconds)*.

Furthermore, consider the processing of ten files using ROTF and NDS. The time to process ten files using ROTF would be equivalent to the time needed to process six files. All five non-root processes could analyze two files each instead of only one process analyzing two files. ROTF could process ten files in the same time it could process twelve files. Two processes would just be idle during the second round of files. As shown in Figures 10 and 12, ROTF takes *1.922 minutes* to process twelve files and NDS takes *2.108 minutes* to process six files. Therefore, the time difference between NDS and ROTF in processing ten files would be approximately *0.186 minutes (11seconds)*.

When comparing ROTF and NDS, ROTF takes the least amount of time to process five, six, ten, twelve, or eighteen files. It has been demonstrated that the time difference between the two algorithms becomes very small when NDS is given an evenly distributed workload. The choice of the best algorithm is not straightforward. If the only consideration in choosing the best algorithm is the timing results, then ROTF is the best choice. However, other considerations, which may be the focus of future research, could

make NDS the best choice. The considerations include the sequence in which abnormalities are found, the impact of file I/O through NFS, and the reassignment of the master template. These issues will be addressed in more detail in Section 5.2.

4.3 Templates Generated

The number of abnormalities and templates found by each parallel algorithm were equivalent in most cases. When processing six or twelve files, the MPCC algorithm found one more abnormality than the other approaches, which led to formation of an extra template. When processing eighteen files, the same implementation found two more abnormalities than the other methods but the same number of templates. The other algorithms produced identical results when processing six or twelve files, but they begin to produce different numbers of templates when processing eighteen files. These differences are attributed to the different sequences in which the implementations are finding abnormalities and averaging them into templates. The differences are not very significant. The number of abnormalities found ranged from 7, when processing six files, to 37, when processing eighteen files. The number of templates formed only ranged from 3 to 6 in most cases, although ROTF did find nine templates once.

The algorithms were tested on data from an animal that was known to have developed a condition called QT elongation prior to its death. Below an illustration is provided in Figure 16 of the master template for the animal and one of the templates that was formed by processing. This template could be representative of the development of QT elongation.

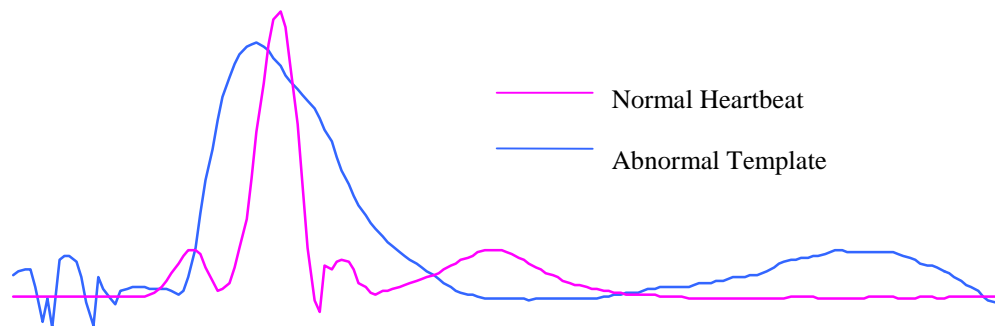


Figure 16: Normal Heartbeat vs.
Abnormal Template

Examples of other templates found are shown in Figures 17 and 18. Templates are formed from any waveforms assigned trigger points that exhibit different morphologies than the master template, which could include noise. The templates shown Figures 17 and 18 could be cardiac events or noise.

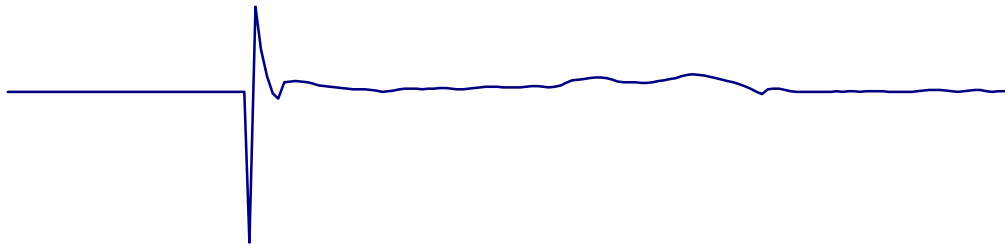


Figure 17: Example of a template that could be noise
or a cardiac event.

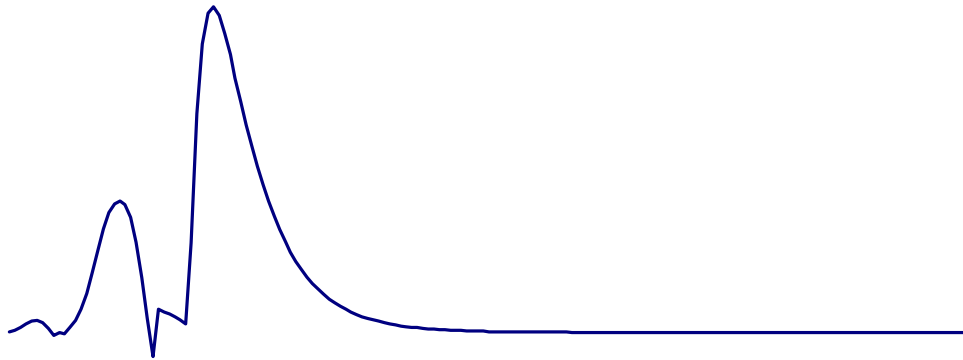


Figure 18: Example of a template formed from a noisy signal

The success of all the approaches is highly dependent upon the number of templates that are found. Very few templates were found in the testing detailed in this chapter. In practical application, the likelihood of a large number of templates for one animal is low. Therefore, the results presented here will most likely remain consistent. If the case of a large template set is considered, the results given above could change. For example, consider the ROTF algorithm. The length of processing is extended by the amount of time needed to combine the abnormalities found by non-root processes into the templates already formed by the root process. If a large number of templates are present, this organization time could greatly reduce the performance of that approach.

Chapter 5

Summary

5.1 Review

The objective of this research was to investigate various methods for parallelizing template-based ECG processing. The need for these types of applications arises from the large amounts of data that are collected from long-term ECG collection. Four alternatives for parallelizing the algorithm that searches ECG data for abnormalities and combines them into a set of templates are presented. The testing of the algorithms was performed on a six-node Linux Beowulf cluster. The performance of the algorithms was evaluated based on the number of processors and the amount of work given to each process.

All but one of the methods discussed did offer significant speedup in the algorithm. The approach known as Root-Only Template Formation (ROTF) gave the best results. When used on six processors, ROTF consistently resulted in a speedup value of at least five. The Node Data Server (NDS) algorithm, however, showed potential for outperforming ROTF given a balanced workload. The performance of the Multiple Process Correlation Computation (MPCC) method becomes independent of the number of processors after 3 processors are utilized. This is because MPCC is dependent on the number of shifts that occur for each computation of the correlation coefficient and the number of templates. Since the shift value is small, increasing the number of processors beyond a certain point is not needed. The Thread Data Server (TDS) approach did not

achieve the results desired. The way in which it was developed may not have been the optimal method.

Based on the results found in this research, we can estimate the amount of time needed to process the 81 Gbytes of data described in Chapter 1. Chapter 1 stated that a 500MHz Pentium III computer could potentially take up to 11 days to process 81 Gbytes of data. This estimate was based on the processing time of one 8 Mbyte file. We can use the data provided in Section 4.1 to estimate the time needed for the six-node Beowulf cluster to process the 81 Gbytes. Since ROTF had the best results, we will use its results for estimation. The estimate is that 81 Gbytes of data could be processed in 26-28 hours using ROTF. This time only includes the abnormality detection stage of the ECG processing software. It does not include the time for pre-processing.

5.2 Future Work

A few topics of this research may be expanded to further explore the concept of distributed template-based ECG processing. One of these topics is the selection of the waveform analysis algorithm. CWA has been shown to be a good method for detecting morphological changes in previous studies. Other algorithms mentioned herein such as BAM, NAD, and DAM could also be explored for their performance in a parallel application. The overall template generation algorithm would not change. Only the method in which a heartbeat window and a template are compared would change. NAD, in particular, was found in other studies to detect occurrences of VT better than CWA. Even though it is vulnerable to baseline fluctuations, it may perform well in distributed template generation.

Another topic that could be explored further is the use of multiple threads on each node. NDS performed very well even though its workload was not always well distributed among processes. TDS did not succeed in improving the performance of the algorithm with the addition of an additional thread on the root process. If TDS were altered in order to eliminate the communication strains, the data-server algorithm could far exceed the performance given by ROTF.

There are several other issues that can affect an algorithm's success. Future study of some of these topics is probably needed. For example, how significant is the sequence in which abnormalities are found and averaged into templates? Is it better for the abnormalities to be averaged into the templates in the order that they occurred? If the sequence is important, the NDS algorithm would not be a good choice for parallelizing the template generation. Furthermore, is it crucial to maintain a count value for the master template? If so, ROTF would not be a good algorithm. Only the root process of that algorithm can update the count of the master template. All of the occurrences of the master template found by non-root processes are not recorded in ROTF.

The performance of all the algorithms presented could improve by reducing the file input/output time encountered by using the NFS protocol. In MPCC, NDS, and TDS, only the root process writes to files. All other processes read the sample and trigger files through the network. In ROTF, the non-root processes write to abnormality location files located on the root machine. Could writing to files over the network greatly impact the processing time? Eliminating the use of NFS and finding an efficient way for all processes to share files with limited network communication could improve all of the algorithms.

5.3 Lessons Learned

Before progress could be made in this thesis research, two obstacles had to be overcome. The first obstacle was learning to administer a Linux Beowulf Cluster with no prior system administration experience. This involved learning more about the Linux environment as well as becoming familiar with system administration concepts. The second obstacle for this research was learning to develop parallel algorithms. A substantial learning curve is encountered when trying to convert a software developer's way of thinking from sequential to parallel design.

One difficult task was learning to correctly synchronize the communication between processes to avoid deadlock. The MPI libraries were very understandable and easy to use. The way in which the libraries handle interprocess communication is convenient which makes learning the concepts of parallel programming easier.

References

1. J. M. Jenkins, S.A. Caswell, "Detection Algorithms in Implantable Cardioverter Defibrillators," *Proceedings of the IEEE*, vol. 84, no.3, pp 428-445, Mar. 1996.
2. "Sudden Cardiac Death",
http://www.americanheart.org/Heart_and_Stroke_A_Z_Guide/sudden.html,
American Heart Association, 2000.
3. R. D. Throne, J. M. Jenkins, L. A. DiCarlo, "A Comparison of Four New Time-Domain Techniques for Discriminating Monomorphic Ventricular Tachycardia from Sinus Rhythm Using Ventricular Waveform Morphology," *IEEE Trans. Biomed. Eng.*, vol. 38, pp.561-570, June 1991; US Pat. No 5,000,189, Mar. 1991.
4. M. Mirowski, M. M. Mower, and P. R. Reid, "The Automatic Implantable Defibrillator," *Amer. Heart J.*, vol. 100. No. 6, pp. 1089-1092, Dec. 1980.
5. D. Lin, L. A. DiCarlo, and J. M. Jenkins, "Identification of Ventricular Tachycardia using Intracavitary ventricular electrograms: Analysis of time and frequency domain patterns," *PACE*. vol. 11. part 1, pp. 1592-1606, Nov. 1988.
6. S. Wakasugi, A. Wada, Y. hasegawa, S. Nakano, N. Shibata, "Detection of Abnormal Cardiac Adrenergic Neuron Activity in Adriamycin-Induced Cardiomyopathy with Iodine-125-metaiodobenzylguanidine," *Journal of Nuclear Medicine*, vol. 33, pp. 208-214, 1992.
7. S. Yakubo, Y. Ozawa, K. Komaki, "Intra-QRS High-Frequency ECG Changes with Ischemia: Is It Possible to Evaluate These Changes Using the Signal-Averaged Holter ECG in Dogs?" *Journal of Electrocardiology*, vol. 28 (suppl), pp.234-238, 1995.
8. C. F. Opitz, G. F. Mitchell, M. A. Pfeffer, J. M. Pfeffer, "Arrhythmias and Death After Coronary Artery Occlusion in the Rat: Continuous Telemetric ECG Monitoring in Conscious, Untethered Rats," *Circulation*, vol. 92, pp. 253-261, 1995.
9. A. P. Vogel, G. P. Jaax, T. M. Tezak-Reidm, S. I. Baskin, J. L. Bartholomew, "Ambulatory Electrocardiography (Holter Monitoring) in Caged Monkeys," *Laboratory Animals*, pp. 16-20, Jan. 1991.
10. "Ambulatory Electrocardiography", *Complete Guide to Medical Tests*,
<http://search1.healthgate.com/tests/test12.shtml>, HealthGate Data Corp. 1999.

11. Y. Watanabe, Y. Nose, M. Yokota, T. Anan, M. Nakamura, "A Holter-Tape Analyzer Employing Circuits for Calculation of Correlation Coefficients," *Medical Informatics*, vol. 12, no. 4, pp. 263-271, 1987.
12. J. H. van Bommel, M. A. Musen, "HandBook of Medical Informatics," pp. 407 - 408, 1997.
13. R. MacDonald, J. Jenkins, R. Arzbaeher, R. Throne, "A Software Trigger For Intracardiac Waveform Detection With Automatic Threshold Adjustment," *Comp Cardiol*, pp. 167-170, 1990.
14. T. L. Sterling, J. Salmon, D. F. Becker, D. F. Savarese, "How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters," pp. 148-153, 1999.
15. W. Gropp, E. Lusk, A. Skjellum, "Using MPI: Portable Parallel Programming with the Message-Passing Interface," pp. 5-19, 202-237 1994.
16. B.K. Grant, A. Skjellum. "The PVM Systems: An In-depth analysis and documenting study", Concise Edition. Technical report UCRL-JC-112016, Lawrence Livermore National Laboratory, August 1992.
17. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. "PVM: Parallel Virtual Machine - A User's guide and Tutorial for Network Parallel Computing. MIT Press, 1994.