Procurement, Benchmarking and Usage of a New High Performance Computing Cluster

by
Gerald R. Busardo

A MASTERS PROJECT
Submitted in partial fulfillment of the requirements
For the degree of Master of Science in
The Department of Computer Science in
The Graduate Program of
Montclair State University

October 2010

MONTCLAIR STATE UNIVERSITY

Procurement, Benchmarking and Usage of a New High Performance Computing Cluster

by

Gerald R. Busardo

A Master's Project Submitted to the Faculty of

Montclair State University

In Partial Fulfillment of the Requirements

For the Degree of

Master of Science

October 2010

School College of Science and Mathematics
Department Computer Science

Acknowledgements

This Master's Project has been one of the most enjoyable and rewarding experiences at Montclair State University, and a fine way to end this chapter of my academic career. What follows is an attempt to give thanks to all that I feel indebted to for providing me with this opportunity, and for supporting me not only throughout this process, but throughout my entire Graduate program.

First and foremost, I would like to thank my Master's Project Advisor, Dr. Stefan Robila, for not only providing me with an opportunity to work with him on a subject as interesting as this, but for his steady support, guidance and inspiration throughout the process.

I also want to thank my Graduate Advisor, Dr. Benham, and the members of the Graduate Committee, Drs. Antoniou, Jenq and Wang for their guidance provided throughout my studies and project work at Montclair State University.

I also would like to express my appreciation for Joseph Youn and his colleagues on Montclair State's CORE team for their technical assistance with the cluster, as well as for providing me with several photos of the cluster.

Additionally, I would like to take this opportunity to acknowledge and give thanks to Daniel Haurey, my friend, professional mentor, and employer over the course of my entire graduate program. Without Dan's constant support, both financially and otherwise, as well as his flexibility and accommodations over the last several years, I likely would not have been able to make it this far.

Lastly, and most importantly, I give thanks for my beautiful wife, Vittoria, for her unwavering friendship, support, love and encouragement for not just this but any and all of life's challenges. The same goes for all of my loving family and friends, for which without them none of this would have been possible, nor worth the effort.

This work was supported by the National Science Foundation's MRI Program under Grant Number CNS-0922644. 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 NSF.

Abstract

Montclair State University received a Major Research Instrumentation grant from the National Science Foundation for a Linux-based High Performance Computing Cluster with (512) 64bit compute cores, 1TB of RAM and 5TB of storage. In this Master's Project paper, we describe my involvement in the procurement process, as well as describe a benchmarking strategy that would cover the three primary layers of any such cluster: the individual node level, the node to node interconnectivity level, and the parallel overlay level. We also present results along with a brief analysis for a variety of benchmarks in order to ensure the system performs as required.

Further, this project involved the design and implementation of an application that could harness the power of this new cluster in order to more efficiently perform best band selection, a fundamental feature extraction problem for Hyperspectral Data Analysis. We start by briefly providing an overview of Hyperspectral Data, its analysis and the concept of best band selection. We then describe an algorithm that performs an exhaustive search for the solution using the distributed, multicore environment and MPI in order to show how using such a solution provides significant improvement over traditional sequential platforms. We conclude with several additional experiments on the robustness of the algorithm in terms of data and job sizes, and a conclusion on the results.

Contents

List of Tables

List of Figures

1 Introduction

Dr. Robila and Montclair State University (MSU) received a grant from the National Science Foundation (NSF) for a Linux-based High Performance Computing Cluster (HPCC) with 512 x86-64bit compute cores. Once delivered, installed and tested, the new HPCC was intended to support computational science research and education at MSU across several different departments, including Computer Science, Mathematics and Linguistics. This Master's Project essentially chronicles my involvement with this new cluster, which started very early on in the procurement phase and over one year ago. The project, as with my involvement, can be broken into two separate but related phases: Pre-Procurement Preparations and Post-Delivery Benchmarking and Parallel Application Development. What follows is a breakdown of each phase:

- **Pre-Procurement Preparations**
  - Independent research and education on High Performance Computing and HPC Clusters, and the Beowulf design,
  - A review of the actual component specifications as listed in the grant proposal,
  - Development of a *Request for Proposal* that was to be provided to various vendors,
  - Research on cluster benchmarking and the development of a comprehensive benchmarking strategy.

- **Post-Delivery Benchmarking and Parallel Application Development**
  - Employing various segments of the aforementioned benchmarking strategy,
  - Testing the capabilities of this new system via the design and implementation of a parallel best band selection (PBBS) application.

2 The Cluster

2.1 Beowulf

A cluster is created when independent computers are combined together to create a single system through special software and networking. For business-environments, cluster configurations are most often used to provide a High Availability (HA) or Failover environment for a critical technology resource. [**1**] An example could be a mail server for a business that heavily relies upon email. In a non-cluster environment, should that mail server fail email will become inaccessible. However, with a cluster configuration, should one mail server fail the other(s) in the cluster can take over its responsibilities in a way that is often automatic and seamless. Clusters are also used for business or science in order to provide High Performance Computing (HPC) resources for computationally intensive tasks, such as weather simulations, financial forecasts or mathematical research. Typically, HPC clusters utilize a Beowulf design, and it is with that design that this project will focus.

A Beowulf Cluster is a scalable computation resource that leverages interconnected commodity hardware, open source software, and message-passing middleware to allow code designed to be executed in parallel to leverage the greater resources a cluster provides. [**1**] The commodity hardware can be anything from a group of everyday single or multiple-core PCs, to many high-end, server-grade and rack mounted units. This hardware often is similar, but does not necessarily have to be (although, benchmarking and other tasks are more difficult with heterogeneous hardware, as will be made evident further down in this paper). The open source software that comprises the Operating System across all nodes typically is a variant of Linux, and the interconnectivity typically is provided via Ethernet, although other methods of interconnectivity can be used. Access to the cluster is typically achieved via a console or remote connection to a single master node, which often is simultaneously connected to an "outside" network as well as a private, cluster-only network. That private network is how the master and the compute nodes will communicate. Each of the compute nodes are dedicated to computation and nothing else, and often are "headless" which is to say that no monitor, mouse or keyboard is connected to allow a user direct control. Figure 2-1 visually depicts such a typical Beowulf design.

2.2 MSU Cluster's Hardware Specification

Since an original cluster configuration and quotation was obtained prior to when the funds would actually be made available, a hardware specification and quotation refresh was necessary. As such, one of my tasks was to create a fresh Request for Quotation (RFQ) and to then send it to several vendors. We then would compare the quotations we receive against an updated quotation from the original vendor before making the ultimate determination on which vendor will provide MSU with the cluster. The resulting RFQ that was created and distributed to the various HPC vendors can be found in Appendix I.

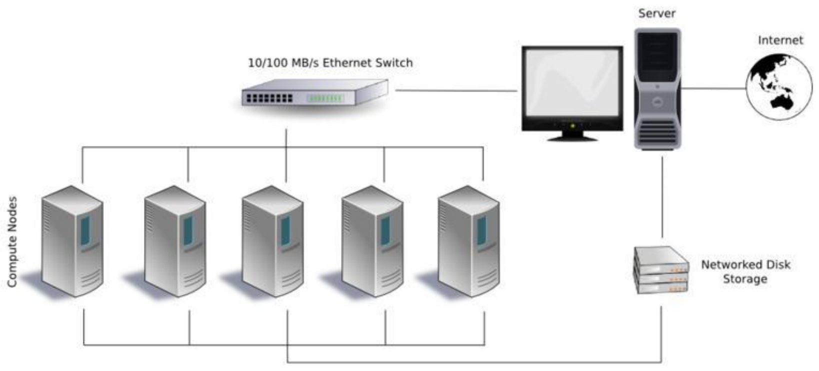


Figure 2-1 - Typical Beowulf Design [1]

2.3 Procurement

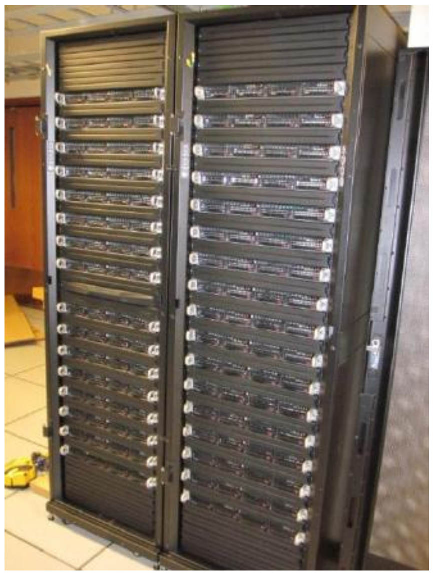
Once the HPC vendor was chosen, the next step was for the University to physically acquire the cluster, and to assemble and integrate it into the University's network environment. A purchase order was made in early December of 2009, and Microway (www.microway.com), the chosen vendor, began work on the configuration soon thereafter. The cluster shipped mid-January of 2010, and was delivered, assembled and configured by the middle of April 2010.

The HPC cluster purchased consists of a master node and (64) compute nodes consolidated into (32) Twin Dual Opteron chassis interconnected via (2) Netgear PROSAFE 48 port Gigabit Stackable Smart Switches. The equipment is enclosed within (2) APC NetShelter SX 42U enclosures.

The master node includes two, quad-core 2.4GHz processors, 8GB of RAM and 500GB of storage. Each of the (64) compute nodes consists of two, quad-core 2.4GHz processors, 16GB of RAM and 50GB of storage. In total, the cluster provides (520) 64-bit, 2.4GHz cores, 1.008TB of RAM and 3.7TB of storage in order to meet or exceed the minimum requested computation power required to achieve a theoretical max of 2.5TFlops.

The Operating System on each node is Red Hat Enterprise Linux 5, with the 2.6.18-164.6.1.e15 (x86_64) Linux kernel. Parallel communications are accomplished via MPICH2 version 1.2, a high-performance implementation of the Message Passing Interface (MPI) standard [**2**]. Compilations would rely on vendor-installed GNU project C and C++ compilers.

From the master node, management and job scheduling is accomplished via Microway Cluster Management Software (MCMS) and Cluster Resources, Inc.'s Maui scheduler, respectively. Students and faculty can access the master node to submit jobs from either an on-campus terminal, or off-campus using any SSH client.


(a)

(b)

Figure 2-2 - a) MSU's new HPC Cluster, b) Dr. Stefan Robila with the new Cluster

3 Cluster Benchmarking

3.1 Benchmarking Metrics

There are numerous methods for measuring system performance (benchmarking), with each method varying greatly in complexity, meaningfulness and accuracy. What follows is a brief discussion of the most widely used methods. [**3**]

3.1.1 Theoretical Peak Performance

A theoretical maximum could be looked at as the *maximum aggregate performance of a system when there is no environmental overhead*. [**3**] This is the performance that often is used when a manufacturer advertises their product, since, as one would clearly expect, such an environment will facilitate the highest possible performance (which makes marketing much easier). Yet, marketing aside, providing such a statistic is a useful way to allow a consumer to quickly perform apples to apples comparison with other similar systems. In developing a method of product evaluation, it would be reasonable to conclude that while one should never rely on theoretical maximums as the only metric to consider when evaluating system performance, one should include it as part of their evaluation methods – especially when comparing with other similar systems.

The method of deterring a theoretical maximum varies depending upon the system being considered. For HPC systems, theoretical peak performance often is defined in floating-point operations per second, calculated by:

$$Rpeak = N * C * F * R \tag{3-1}$$

where *Tpeak* is the aggregate performance, *N* is the number of nodes, *C* is the number of CPUs per node, *F* is the number of floating-point operations per clock period, and *R* is the clock rate (measured in cycles per second). [**3**] With the current performance of modern computing clusters, *Tpeak* more often is given in millions of floating-point operations per second (Mflop/s - megaflops), billions of floating-point operations per second (Gflop/s - gigaflops), trillions of floating-point operations per second (Tflop/s - teraflops) or even in a thousand-trillion floating-point operations per second (Pflop/s - petaflops).

3.1.2 Percentage of Peak

Percentage of Peak is simply the *percentage of Actual Performance* (*Rmax) to Theoretical Peak Performance* (*Rpeak*): [**3**]

$$P = \frac{Rmax}{Rpeak} \tag{3-2}$$

Using the same example as with Theoretical Peak performance, Percentage of Peak would be the actual floating-point operations per second result of a benchmark, divided by the Theoretical Peak performance in floating-point operations per second. Such a percentage is helpful when tuning a system to ensure that the final benchmark result is beginning to arrive as close as it possibly could to peak. This is achieved by comparing the Percentage of Peak to other, known Percentage of Peak results for similar and what is collectively considered to be fine-tuned systems.

3.1.4 Application Performance

Application performance is *the number of "operations" performed by an application, divided by the total run time*. [**3**] This could be looked at in two different ways. The first approach is to literally determine, fairly accurately, the number of floating-point (or integer) operations actually performed by the code. With this method, the final measurement would be in Mflop/s, Gflop/s, (or Mop/s, Gop/s, if integer). However, such a determination is not always easy. The second approach is simply to consider the number of completed operations. This second method is only useful if the code has a simple purpose. An example would be the application created as part of this project (see section 4 below) in which application performance could be the number of intervals computed for best band selection in a given period of time. Regardless of the approach, such a metric is typically much more meaningful than a theoretical peak measurement if a user ultimately intends on running the application being used to gauge performance.

A note on Application Performance as a benchmarking metric: inefficient code, as well as a poorly tuned system will greatly affect the final measurement and ultimately prohibit one from obtaining a true performance measurement of a given system.

3.1.5 Application Run Time

Application Run Time simply means *the wall-clock run time for a given application*. [**3**] This metric is obviously much easier to calculate than Application Performance since no analysis of the code is necessary. As with Application performance, Run Time as a metric also falls victim to a poorly tuned system as well as inefficient coding.

3.1.6 Scalability

Scalability is a performance metric that measures *an algorithm's (or application's) ability to benefit from a parallel architecture*. [**3**] While not a means to measure performance of a hardware system, scalability will often be used when working on high performance computing clusters, as we will do as well later in this paper (see 4.4.2.2 below).

Scalability often is computed as: [**3**]

$$S = \frac{T(1)}{T(N)} \qquad (3\text{-}3)$$

where $T(1)$ represents the Application Run Time metric with one node, processor or compute core, and $T(N)$ represents Application Run Time with N nodes, processors or compute cores. A scalability result close to N means that the algorithm or application performs well in a parallel environment (i.e. a linear speedup has been achieved).

3.1.7 Latency and Bandwidth

Latency (time delay) and bandwidth (transfer rate) are metrics that can measure node to node interconnectivity performance. [**3**] Such measurements are often given in seconds and bits per second, respectively.

3.2 The Benchmarking Strategy

In the Information Technology field, particularly with the technical task of supporting and troubleshooting systems, applications or networks, a common and best practice first step to any complex troubleshooting effort is to break a system down into its constituent parts and handle each individually before tackling the big picture issue. By doing so, you not only gain insight into the role of each individual component, but you gain a better understanding of the overall system in a way that is sometimes impossible without witnessing the individual components operating individually.

This type of "functional decomposition" approach is also beneficial in devising a strategy for HPCC benchmarking since, at first, a "computer" with dozens, hundreds or even thousands of cores, nodes, wires, hard drives and sticks of RAM seems impossibly complex. Instead, one can consider a cluster to consist of three separate layers: [**4**]

- Layer 1: The Individual Node
- Layer 2: Node to Node Interconnectivity
- Layer 3: Parallel Processing Overlay

In a perfect situation, benchmarking system performance would be performed on each of these layers. What follows are what we feel would be the most relevant metrics and methods for benchmarking the new HPC cluster for each of the aforementioned layers and many of the aforementioned metrics.

3.2.2 Layer One: The Individual Node

At this layer, the primary focus should be on the individual OS, and the immediate hardware supporting it - no focus should be on the interconnects between nodes, or the MPI libraries installed to facilitate parallel instructions. This is the layer where hardware burn-in testing is most valuable.

Often, HPC cluster vendors would provide their own proprietary tools for such burn-in testing at the Individual Node Layer. In our case, Microway provided us with the MPI Diagnostic Suite. This suite contains tools that can measure the bandwidth and latency between all nodes in a cluster and then summarize the data graphically to spot problem nodes. In particular, the MPI Link-Checker diagnostic can detect issues with processer caching, motherboards, PCI bus, BIOS, et cetera.

Other examples of benchmarks and diagnostics that could be run at this layer would include:

- Disk performance benchmarks
  - Bonnie++ [**5**]
    - A remake of the Bonnie benchmark created by Tim Bray and consists of several separate tests related to file IO and file creation.
  - IOzone [**6**]
    - Generates and measures a variety of file operations including: read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread ,mmap, aio_read and aio_write.
- CPU performance benchmarks
  - DGEMM [**7**]
    - This benchmark, which stands for General Matrix Multiply, reports on the floating point performance of a compute core.
    - General Matrix Multiply is a subroutine in the Basic Linear Algebra subprograms (BLAS). These subprograms perform matrix multiplication. The D in DGEMM stands for double-precision.
- Memory performance benchmarks:
  - Streams [**8**]
    - Memory performance could be measured by the Streams benchmark, which measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for four simple vector kernels which form the basis of many long vector operations.
    - Table 3-1 describes the four vector kernels along with the per-iteration bytes and FLOPS.

		Per iteration:	
Name	**kernel**	**Bytes**	**FLOPS**
COPY	a(i) = b(i)	16	0
SCALE	a(i) = q*b(i)	16	1
SUM	a(i) = b(i) + c(i)	24	1
TRIAD	a(i) = b(i) + q*c(i)	24	2

Table 3-1 - STREAMS Operations

3.2.3 Layer Two: Node to Node Interconnectivity

Benchmarking and diagnostics at the interconnectivity layer should focus on the hardware that connects each node to the other, including network cards (NICs), Ethernet cables and Ethernet switches. The goal will be to ensure that measured network bandwidth matches that of which the technical specifications advertise, and that there is minimal latency. Examples of benchmarks and diagnostics that could be run at this layer:

- Network Performance Testing via Netperf
    - Netperf's primary testing and benchmarking focus is with regard to unidirectional data transfers and request/response performance over TCP or UDP [**9**]
    - Once a server daemon is started on a remote system, one would invoke a client program on a local system in order to begin initiating one of Netperf's tests. Once the client program begins to run, it first establishes a control connection to the server component on the remote system. Once established, a separate data connection is opened for the measurements to be performed. Once the tests are completed, the data connection is closed and results from the server component are passed back to the client via the control connection.

3.2.4 Layer Three: Parallel Processing Overlay

At this top layer, parallel compute jobs rely on special overlays such as Message Passing Interface (MPI) in order for many computers to communicate with one another to achieve a common goal [**10**]. Testing this overlay also, indirectly, ensures that the bottom two levels are working synergistically in order to perform calculations at supercomputer rates. This is the level that real-world applications or problems can and should be a part of the benchmarking so that we can see how the system will work once it becomes available to the university. Examples of benchmarks and diagnostics that could be run at this layer:

- Floating Point Operations Benchmark: HP Linpack [**11**] [**12**]
    - When one thinks of benchmarking, in general, and especially on a High Performance Computing Cluster, they often think of the Top500. The Top500 is a list of the 500 most powerful (i.e. fastest) computer systems, compiled twice yearly since June 1993. Within the list, supercomputers and clusters are ranked using the High Performance Linpack (HPL) benchmark.
    - The HPL benchmark is a software package that provides a "testing and timing program to quantify the accuracy of the obtained solution as well as the time it took to compute it." In particular, it "solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers."
    - To run this benchmark, one must have a working Message Passing Interface (MPI), as well as either the Basic Linear Algebra Subprograms (BLAS) or Vector Signal Image Processing Library (VSIPL), whose libraries are required in order to perform the algorithm's computations. [**13**] Further, these libraries should ideally be optimized for the particular system the benchmark is to be performed on; otherwise, performance will suffer. [**14**]

- An advantage of the Linpack benchmark is that the calculation of performance once a timing figure is obtained is very easy. Further, there is a huge collection of results with which one can compare in order to determine how one's system stacks up against other similar systems. [**3**]
- A principal disadvantage of the Linpack benchmark is that the results tend to over-estimate the performance that real-world scientific applications can expect to achieve on a given system. It is not uncommon for the Linpack benchmark, for example, to achieve a very high Percentage of Peak (i.e. 30% or more of the theoretical peak performance potential of a system). Real scientific applications, in contrast, seldom achieve more than 10% of Theoretical Peak Performance. [**3**]

- Real Scientific Application: NAS Parallel Benchmarks
    - The Numerical Aerodynamic Simulation (NAS) Program, located at NASA Ames Research Center, created the NAS Parallel Benchmarks (NPB) in order to objectively measure the performance of various highly parallel computer systems and to compare them with conventional supercomputers. The NPB consists of a set of eight problems, each of which focuses on some important aspect of highly parallel supercomputing for aero-physics applications. [**15**]
        - The Embarrassingly Parallel (EP) Benchmark [**15**]
            - In this benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudorandom numbers. This benchmark provides an estimate of the upper achievable limits for floating-point performance on a particular system.
        - Multigrid (MG) Benchmark [**15**]
            - The second benchmark is a simplified multigrid kernel, which solves a 3-D Poisson Partial Differential Equation (PDE). To simulate more of a real-world type of application, this problem is simplified to have constant rather than variable coefficients. The goal of this benchmark is to test both short and long distance highly structured communication.
        - Conjugate Gradient (CG) Benchmark [**15**]
            - In this benchmark, "a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix." Typical of unstructured grid computations, this benchmark tests irregular long-distance communication and employs sparse matrix-vector multiplication.
        - FFT PDE (FT) Benchmark [**15**]
            - In this benchmark a 3-D partial differential equation is solved using fast Fourier transforms. The goal is to obtain a good test of long-distance communication performance.
        - Integer Sort (IS) benchmark [**15**]
            - This benchmark tests a sorting operation that is used to reassign particles to the appropriate cells, such as often used in particle method coding and similar to "particle-in-cell" applications of physics, where particles or fluid elements are assigned to cells and may drift out. The goal of this benchmark is to test both integer computation speed and communication performance.
        - Simulated Computational Fluid Dynamics (CFD) Benchmarks [**15**]
            - The first of the three simulated CFD applications is the lower-upper diagonal (LU) benchmark. This benchmark "employs a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block *5x5* lower and upper triangular system".
            - The second of the three simulated CFD applications is the scalar pentadiagonal (SP) benchmark. In this benchmark, "multiple independent systems of nondiagonally dominant, scalar pentadiagonal equations are solved".
            - The third and final simulated CFD application is the block tridiagonal (BT) benchmark. In this benchmark, "multiple independent systems of non-diagonally dominant, block tridiagonal equations with a *5x5* block size are solved". The primary difference between the scalar pentadiagonal (SP) and block tridiagonal (BT) benchmarks is with respect to the

communication to computation ratio, with the BT benchmark requiring more computation than its counterpart.

Each NAS Parallel Benchmark listed above can be executed using different problem sizes, defined by a “class” definition. For almost all benchmarks, the following class sizes exist (in increasing problem size): A, B, C and D. Integer Sort (IS) does not have a class “D” problem size.

Unlike the HPL benchmark, the NAS Parallel Benchmark suite allows one to witness performance on a real-world, scientific application, which often is much less than Theoretical Peak Performance. [**3**] Another advantage of this benchmark is that, as with the HPL benchmark, it can be used to measure performance of a single node Beowulf System, as well as an entire Beowulf system. This is made possible via different, pre-defined problem sizes. For example: the “sample” or “Class W” size problems can be easily run on a single-processor system. For a cluster with 32 processors, the Class A size is an appropriate test. The Class B problems are more appropriate for systems with roughly 32-128 processors. And, the Class C problems can be used for systems with up to 256+ CPUs. [**3**]

A disadvantage is that, unlike the HPL, measuring performance for purposes of comparison appears to be more difficult with this benchmark. Should you want to compare your results with others, no readily available repositories of system performance seem to exist. However, this benchmark can be used to record baseline performance, and then run again after system changes were made in order to observe any related changes in performance.

3.3 The Benchmarking Results

Due to time constraints, we could not perform all benchmarks from the aforementioned strategy. What follows are the benchmarks chosen for each of the aforementioned levels, along with the results.

3.3.1 Benchmarking Results for Layer One

Prior to the delivery of the cluster, Microway performed various tests in order to ensure that the cluster, as a whole, and each individual node, are operating as they should. In particular:

- Each node was network booted to execute low-level memory tests for 12+ hours
- Each node’s Operating Systems were loaded and Linux stress tests were executed for 24+ hours
  - Stress tests included processor and memory intensive applications that have been shown to cause faults in the field.
  - Stress tests also were sure to access all sectors on all hard drives, as well as to include various filesystem-intensive applications in order to ensure drive and filesystem reliability.

Due to this, and with time limitations in mind, it was decided that individual node benchmarks and tests were not in this project’s best interests. Instead, we could rely on parallel runs to ensure each node is operating as it should, as well as serial runs during the Application phase of this project to ensure individual nodes are performing as they should.

3.3.2 Benchmarking Results for Layer Two

To test the Node to Node Interconnectivity, we used Netperf. What follows is a description of the installation process, and then the various tests that were run.

3.3.2.2 Netperf Installation

The installation package was downloaded directly from the public Netperf Hompage. [**9**] The tar-gzip archive was then unpacked and a configure script was run that automatically guesses the correct values for various system-dependent variables used during compilation. These values are then used to create a 'Makefile' within each directory of the package. Once this configuration process is complete, the package is compiled using the GNU C compiler (gcc) and then installed via 'make install'.

3.3.2.3 Netperf Execution, Results and Analysis

Once Netperf was installed, two executable files are created in the *netperf/bin* subdirectory. The first is *netserver*, which listens for connections from the Netperf client via a specific port and then responds accordingly. The second is *netperf*, which is the Netperf client.

We ran *netserver* on the cluster's master node with no run-time parameters, which loads the server component and causes it to listen via the default port of 12865. We then ran, from each node we wished to test, the following commands:

- To perform a TCP stream test from the master to a slave node for a duration of 10 seconds and with 256 Kbyte socket and buffer sizes (send message size of 262144000) via the standard port:

```
./netperf -H master -l 10 -- -m 256K,256k -M 256k,256K -s 256k,256K -S 256K,256K
```

- To perform the same test but with 1024 Kbyte socket and buffer sizes (send message size of 1047576000):

```
./netperf –H master –I 10 -- -m 1024K,1024K –M 1024K,1024K –s 1024K,1024K –S 1024K,1024K
```

- To perform the same 1024 Kbyte test but with a longer duration (300 seconds):

```
./netperf –H master –I 300 -- -m 1024K,1024K –M 1024K,1024K –s 1024K,1024K –S 1024K,1024K
```

from master to slave #	send message size (in bytes)	throughput (in 10^6 bits/sec)	Elapsed Time (in seconds)
2	262144000	938.10	10
2	1047576000	938.14	10
2	1047576000	938.11	300
16	262144999	939.95	10
16	1048576000	939.31	10
16	1048576000	940.00	300
64	262144999	938.78	10
64	1048576000	938.77	10
64	1048576000	938.68	300

Table 3-2 - Throughput benchmarks via Netperf test runs from various nodes to the master node

Since there were many nodes using the same network medium, and due to time constraints, we chose to only perform this test from three separate slave nodes. The goal for each of these tests was to ensure that the throughput is close to the theoretical max for the network connectivity medium. Sub-optimal throughput would indicate that packet loss or other issues exist. In our case, the cluster is relying upon Gigabit Ethernet, which is capable of 1,000 megabits per second, or 125 megabytes per second. [**16**]

As the results in Table 3-2 indicate, we were able to achieve an average of *939,000,000 bits per second*, or *117 megabytes per second*, regardless if the tests ran for 10 seconds or 300 seconds. Throughput which is 94% of the theoretical max for Gigabit Ethernet falls within an acceptable range and indicates that the network connectivity between nodes performs as one would expect.

3.3.3 Benchmarking Results for Layer Three

We chose to perform each of the benchmarks suggested in our aforementioned benchmarking strategy for this layer since acceptable performance would indicate that the foundational levels are performing as they should.

3.3.3.1 HP Linpack Installation

To test the power of the parallel environment, we started with the popular performance computing benchmark, the High Performance Linpack (HPL). Since the installation of the HPL benchmark required several prerequisite libraries, one of which must be fine-tuned for the particular architecture in order to receive acceptable performance, the overall installation process was more time-consuming and delicate than with Netperf. As such, a detailed description of the installation is provided within Appendix VI –Tuning of the HPL Benchmark. What follows is a description and an analysis of the various tests that were run.

3.3.3.2 HP Linpack Execution, Results and Analysis

The HP Linpack benchmark requires its parameters to be defined in a text file called HPL.dat. A detailed explanation of HPL.dat, as well as the HPL.dat files used in each of our benchmark runs can be found in Appendix VI –Tuning of the HPL Benchmark.

Once HPL.dat contains the necessary parameters for a particular HPL benchmarking run, one would run the HPL executable without any input. However, in order to run the HPL executable on multiple nodes using MPI, one must precede and follow the HPL executable with various commands. What follows are the commands that were used during our benchmarking. A detailed explanation on these commands and the general usage of MPI are provided in Appendix III – Leveraging MPI for Parallelization of C/C++ Code.

- To run the HPL on all 64 compute nodes with each nodes' core being utilized, and with output placed in a text file:

```
mpirun -machinefile computeNodes -1 -np 512 ./xhpl > output-512-2 < /dev/null &
```

- To run the HPL on all 65 compute nodes with each nodes' core being utilized, and with output placed in a text file:

```
mpirun -machinefile allNodes -np 520 ./xhpl > output-520-6 < /dev/null &
```

Nodes	Total Cores	Total Memory (mb)	Swap Needed?	RAM per Core	Problem Size (N)	Partition Blocking Factor (NB)	Process Rows (P)	Process Columns (Q)	Total Time (in seconds)	Gflops
64	512	1,048,576	no	2,048	331520	128	16	32	15017.37	1618.00
65	520	1,056,768	no	2,032	330240	128	20	26	15071.98	1593.00
65	520	1,064,960	yes	2,048	236288	128	20	26	9115.96	964.80
65	520	1,064,960	yes	2,048	100000	256	20	26	1387.45	480.50

Table 3-3 - HP Linpack Results for Varying Parameters

The goal for the HPL benchmark runs were to try and achieve as close to Theoretical Peak Performance of floating-point operations per second as is possible, and ideally, a Percentage of Peak that is similar to that which is achieved by other high performance computing systems of similar design as per the Top 500 list. To calculate Percentage of Peak, we must consider the specifications of Montclair State's cluster and calculate the cluster's Theoretical Peak Performance, or Rpeak. Each node has two, quad-core 2.4 Ghz processors, and there are (64) compute nodes and (1) master node. In total, we have (512) compute cores, and (520) total cores. Further, the type of core that was used has (2) floating point units. With that said, and as per the aforementioned definition of Theoretical Peak Performance, our Rpeak would be:

$$Rpeak = N * C * F * R = 64 * 8 * 2 * 2.4 = \ 2{,}457.6 \text{ Gflop/s } \ or \ 2.5 \text{ Tflop/s} \qquad (3\text{-}4)$$

Regarding Percentage of Peak, the Top 500 fastest supercomputers as of June 2010 had an average Percentage of Peak of 67.14% (see Appendix VII – Top 500 List for June, 2010). Therefore, our goal would be to get as close to 67% of 2.5 Tflop/s in the HP Linpack benchmark.

We ran the HPL a total of four times: two times with just the compute nodes, and two with all nodes. The results, as well as the varying parameters as were defined in the HPL.dat for each run can be found in Table 3-3.

Over the course of the four executions of the HPL benchmark, the highest result we received was 1.6 Tflop/s. In order to see if this result seems reasonable and acceptable, we calculated our Percentage of Peak as:

$$P = \frac{Rmax}{Rpeak} = \frac{1618 \text{ Gflop/s}}{2458 \text{ Gflops/s}} \ = \ 66\% \qquad (3\text{-}5)$$

The resulting Percentage of Peak was only a single percentage shy of the average for the world's top supercomputers. With some additional HPL tuning within the HPL.dat file, there is a chance we can meet or even beat the average Percentage of Peak. However, the closeness of the percentage is enough to show us that our cluster seems to be performing as it should.

3.3.3.3 NAS Parallel Benchmarks Installation

The NAS Parallel Benchmarks version 3.2 gzipped archive was provided by Microway. Within this archive, source code for several different variations of the benchmarks are provided in separate subdirectories, including:

- /NPB3.2-MPI
    - MPI (for Fortran and C)
- /NPB3.2-OMP
    - OpenMPI
- /NPB3.2-SER

    - Serial (non-parallel, for Fortran and C)

Within each subdirectory, documentation, configuration files and source code is provided via the following file structure:

- /bin
    - Where the executable benchmarks are placed after compilation
- Subdirectories containing source code for each benchmark within the suite
    - /CG
    - /EP
    - /IS
    - /BT
    - /DT
    - /FT
    - /LU
    - /MG
    - /SP
- /common
    - Source code and headers shared by each of the benchmarks
- /sys
    - Utilities and files used during the build process
- /config
    - Configuration files that direct the build process to create the chosen benchmarks
- /MPI_dummy
    - An MPI library provided as a convenience for those that do not have an MPI library installed, but would like to try running the benchmarks on one processor.

Once the individual benchmarks one wishes to execute are chosen (SP, LU, MG, etc), one must go within the /config directory and create a *make.def* file. This file provides environmental details to the build process so that the appropriate compilers are found, and used properly. Once *make.def* is created, to compile the chosen benchmarks one would enter this command:

```
make <benchmark-name> NPROCS=<number> CLASS=<class> [SUBTYPE=<type>]
```

where <benchmark-name> is "bt", "cg", "ep", "ft", "is", "lu", "mg", or "sp", <number> is the number of processes and <class> is "S", "W", "A", "B", "C", or "D". If compiling the BT benchmark for I/O testing, one must define <type> as either “full”, “simple”, “fortran”, or “epio”.

On the other hand, if one chooses to compile many benchmarks at once, the configuration parameters mentioned above can be combined in a single *suite.def* file. Then, one would enter this command to compile all at once:

```
make suite
```

A copy of both the *make.def* and *suite.def* files used for this project are contained within Appendix VIIII – NAS Parallel Benchmark Suite; Technical Details.

3.3.3.4 NAS Parallel Benchmarks Execution, Results and Analysis

We chose three of the several available benchmarks within the NAS Parallel Benchmarks suite: IS, EP, LU. For each, we chose different problem sizes so that we could attempt to observe how problem size effects performance, and in an attempt to obtain the highest performance result possible.

The first benchmark, Integer Sort (IS) was executed on all compute nodes of the cluster (512 cores) with three different problem sizes, as is displayed in Table 3-4. As would be expected, Application Performance measured in millions of key ranking operations per second decreased as the problem size increased. Also as expected, Application Run Time increased as the problem size increased.

The second benchmark we ran was the Embarrassingly Parallel (EP) benchmark. As with the IS benchmark, this benchmark was run on all compute nodes. However, unlike the IS benchmark, an even larger problem size via a "D" class is available. The results can be found in Table 3-5. Like the IS benchmark, Application Run Time increased as the problem size increased. However, unlike the IS benchmark, Application Performance measured in millions of random numbers generated per second increased as the problem size increased. This may be a result of benchmarks ability to leverage a parallel computing environment for optimal results, as is indicated by the benchmarks name, or it is a result of smaller problem sizes simply not being optimal for a large cluster.

Since the EP benchmark is one that should be very scalable in a parallel environment, as indicated by its name, we chose to observe the benchmarks results as we varied the number of processes. Table 3-6 shows that as the number of processes double from 1 up to 512 (all compute nodes), Application Performance increases and Application Run Time decreases. A graph representing the speedup from 1 core can be found in Figure 3-1. This is exactly what one would expect with a benchmark that is "embarrassingly parallel". Interestingly, however, Application Performance and Application Run Time decreased when we ran the EP benchmark on the full cluster (520 cores). This likely is due to decreased amount of available RAM on the master node, as well as the increased CPU activity on the master node due to cluster management responsibilities.

type	class	size	iterations	num processes	time (seconds)	mop/s total	Type of operation
IS	A	8388608	10	512	20.93	4.01	keys ranked
IS	B	33554432	10	512	95.1	3.53	keys ranked
IS	C	134217728	10	512	464.47	2.89	keys ranked

Table 3-4 - NPB "IS" Benchmark for varying problem sizes

type	class	size	num processes	time (seconds)	mop/s total	Type of operation
EP	A	536870912	512	0.11	4834.93	random numbers generated
EP	B	2147483648	512	0.3	7233.22	random numbers generated
EP	C	8589934592	512	1.03	8314.77	random numbers generated
EP	D	1.37439E+11	512	15	9163.43	random numbers generated

Table 3-5 - NPB "EP" Benchmark for varying problem sizes

type	class	size	num processes	time (seconds)	mop/s total	Type of operation
EP	B	2147483648	1	118.56	18.11	random numbers generated
EP	B	2147483648	2	59.67	35.99	random numbers generated
EP	B	2147483648	4	29.83	72.00	random numbers generated
EP	B	2147483648	8	14.95	143.66	random numbers generated
EP	B	2147483648	16	7.51	285.99	random numbers generated
EP	B	2147483648	32	3.77	569.59	random numbers generated
EP	B	2147483648	64	1.93	1113.58	random numbers generated
EP	B	2147483648	128	0.97	2222.58	random numbers generated
EP	B	2147483648	256	0.50	4309.90	random numbers generated
EP	B	2147483648	512	0.28	7646.89	random numbers generated
EP	B	2147483648	520	0.36	5936.36	random numbers generated

Table 3-6 - NPB "EP" Benchmark - Scalability Analysis

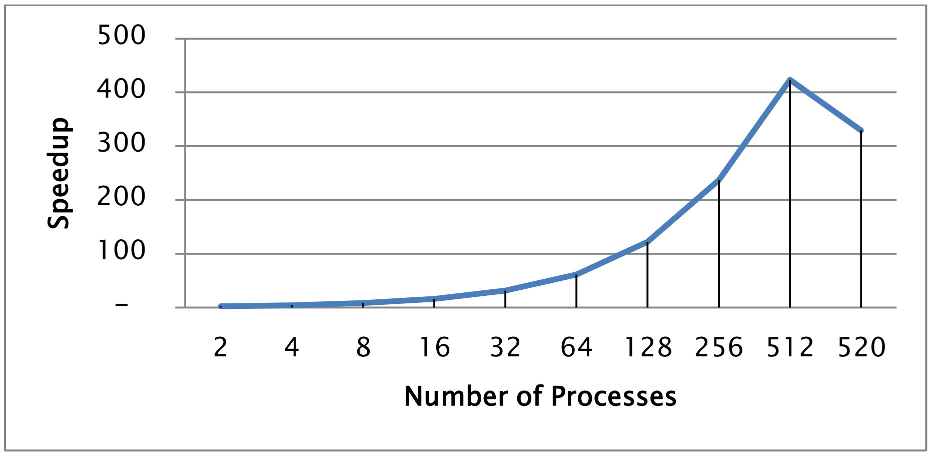


Figure 3-1 - Speedup analysis on NPB's "EP" Benchmark

The third benchmark we ran from the NPB suite was the Conjugate Gradient (CG) benchmark. The results of running this benchmark on four different problem sizes across all compute nodes of the cluster are displayed in Table 3-7. Similarly to all other benchmarks run from the NPB suite, Application Run Time increases as the problem size increases. Unlike the other benchmarks, however, Application Performance seemed to vary depending upon problem size, with a decrease from class "A" to classes "B" and C", and then a peak performance at class "D".

The fourth and final benchmark we ran from the NPB suite was the Lower-Upper symmetric Gauss-Seidel (LU) benchmark. With this benchmark, we chose to see how each problem size scaled as the number of processes doubled from 1 to 256 (for problem sizes "A", detailed in Table 3-8, and "B", detailed in Table 3-9), 32 to 256 (for problem size "C", detailed in Table 3-10), and 64 to 512 (for the largest available problem size "D", detailed in Table 3-11).

For each problem size, the LU benchmark seemed to show a steady increase in Application Performance, defined by millions of floating point operations per second, as the number of total processors increased up to a certain point. At that point, Application Performance would then begin to decrease. Our interpretation of this result is simply that Application Performance will eventually degrade once the benefits of parallelization of a given problem and problem size are less than the overhead caused by the necessary internode communications required by said parallelization.

type	class	size	iterations	num processes	time (seconds)	mop/s total	Type of operation
CG	A	14000	15	512	2.42	617.37	floating point
CG	B	75000	75	512	461.45	118.56	floating point
CG	C	150000	75	512	533.34	268.77	floating point
CG	D	1500000	100	512	2114.96	1722.44	floating point

Table 3-7- NPB's "CG" Benchmark for varying problem sizes

type	class	size	iterations	num processes	time (seconds)	mop/s total	Type of operation
LU	A	64x64x64	250	1	112.03	1064.85	floating point
LU	A	64x64x64	250	2	57.66	2068.88	floating point
LU	A	64x64x64	250	4	29.22	4083.12	floating point
LU	A	64x64x64	250	8	21.83	5465.68	floating point
LU	A	64x64x64	250	16	8.78	13585.45	floating point
LU	A	64x64x64	250	32	7.04	16936.47	floating point
LU	A	64x64x64	250	64	4.08	29223.66	floating point
LU	A	64x64x64	250	128	10.56	11299.56	floating point
LU	A	64x64x64	250	256	36.56	3263	floating point

Table 3-8 - NPB "LU" Benchmark for problem size "A" with varying # of processors

type	class	size	iterations	num processes	time (seconds)	mop/s total	Type of operation
LU	B	102x102x102	250	1	563.43	885.33	floating point
LU	B	102x102x102	250	2	239.13	2085.98	floating point
LU	B	102x102x102	250	4	128.11	3893.85	floating point
LU	B	102x102x102	250	8	69.2	7207.97	floating point
LU	B	102x102x102	250	16	36.73	13580.18	floating point
LU	B	102x102x102	250	32	34.18	14592.12	floating point
LU	B	102x102x102	250	64	34.31	14537.28	floating point
LU	B	102x102x102	250	128	35.15	14192.54	floating point
LU	B	102x102x102	250	256	72.32	6897.26	floating point

Table 3-9 - NPB "LU" Benchmark for problem size "B" with varying # of processors

type	class	size	iterations	num processes	time (seconds)	mop/s total	Type of operation
LU	C	162x162x162	250	32	83.84	24318.68	floating point
LU	C	162x162x162	250	64	50.25	40575.47	floating point
LU	C	162x162x162	250	128	76.77	26558.21	floating point
LU	C	162x162x162	250	256	98.68	20662.27	floating point

Table 3-10 - NPB "LU" Benchmark for problem size "C" with varying # of processors

type	class	size	iterations	num processes	time (seconds)	mop/s total	Type of operation
LU	D	408x408x408	300	64	690.84	57552.76	floating point
LU	D	408x408x408	300	128	469.64	84954.97	floating point
LU	D	408x408x408	300	256	379.89	105026.31	floating point
LU	D	408x408x408	300	512	514.43	77556.87	floating point

Table 3-11 - NPB "LU" Benchmark for problem size "D" with varying # of processors

Figure 3-2 shows the Application Performance increase as we run the LU benchmark on all compute nodes but with increasing problem sizes. This result is similar to each other benchmark we ran from the NPB suite, excluding the IS benchmark, and we believe is a result of both finding the most appropriate problem size to match the capabilities of the cluster, and ensuring that the problem size is not so small that communications overhead begins to significantly impair overall performance.

It is said that real scientific applications, in contrast to benchmarks like the Linpack which have highly favorable data locality characteristics, seldom achieve more than 10% of the Rpeak for modern distributed memory, parallel systems such as Beowulf clusters. [**3**] To see if that is true, we took the Application Performance results of the LU benchmark, in terms of millions of floating point operations per second, and compared them with the Rpeak of our Linux cluster, as defined previously, and the Rmax as achieved via the Linpack benchmarks discussed previously and as summarized in Table 3-13. This comparison is displayed in Table 3-12. Once the most appropriate problem size was found for our sized cluster, which in this case was the largest available, class “D”, we only were able to achieve 4% of the Theoretical Peak Performance (Rpeak) and just about 6.5% of the best Linpack result achieved during our benchmarking.

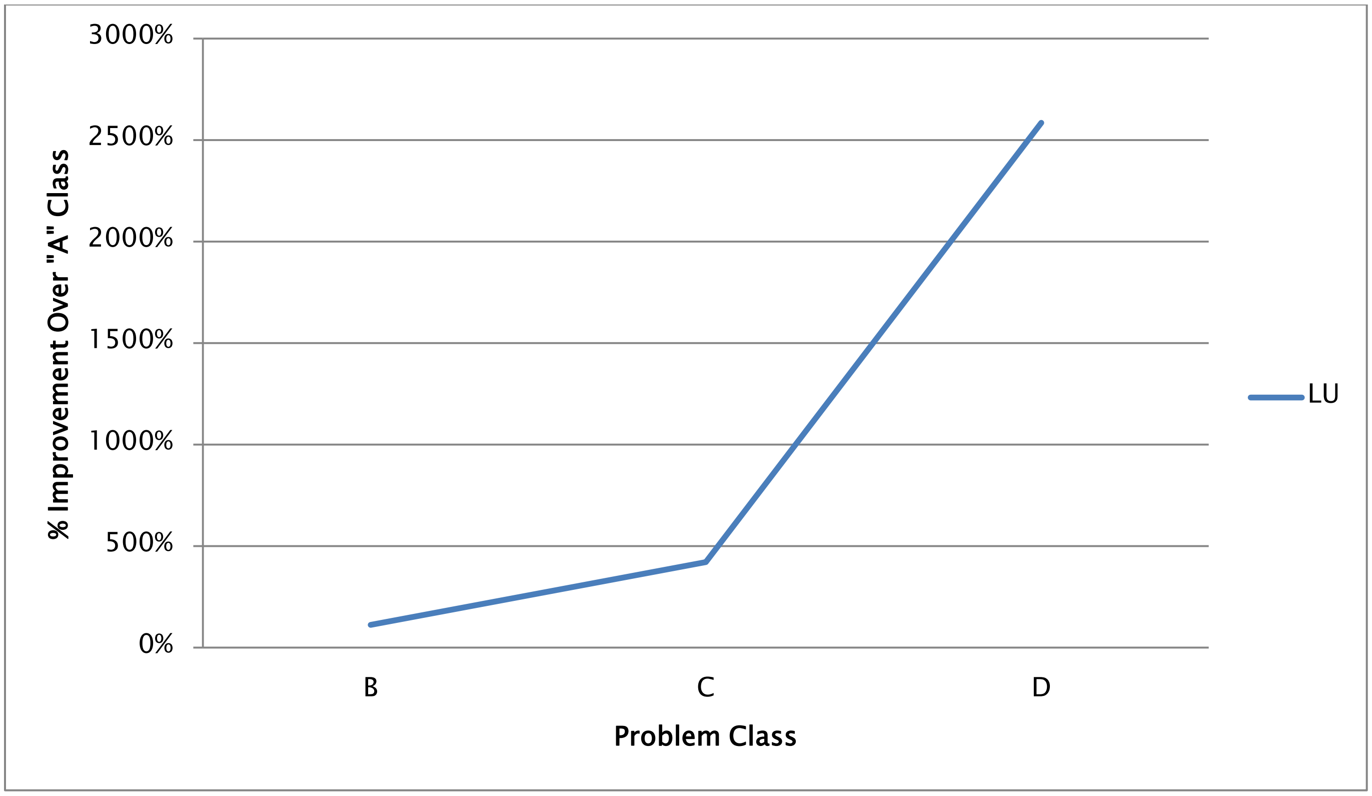


Figure 3-2 - Finding the Optimal Problem Size for NPB "LU" Benchmark

type	class	size	mop/s total	% of Rpeak	% of Linpack Rmax
LU	A	64x64x64	3263	0.13%	0.20%
LU	B	102x102x102	6897.26	0.28%	0.43%
LU	C	162x162x162	20662.27	0.84%	1.28%
LU	D	408x408x408	105026.31	4.27%	6.49%

Table 3-12 - Application Performance of NPB's "LU" Benchmark When Compared to Linpack Rmax and Rpeak

# nodes	64
# cores per node	8
# fpu's per core	2
clock speed (Hz)	2,400,000,000
R_{Peak} (FLOP/s)	2,457,600,000,000
R_{Max} (FLOP/s)	1,618,000,000,000

Table 3-13 - Summarization of the Cluster's Characteristics, Theoretical Peak Performance and actual Linpack Performance

4 Small Application Development: Parallel Best Band Selection

4.1 Introduction: A Brief History on Remote Sensing and Spectral Imaging

Remote sensing, which is the acquisition of information of an object, scene or phenomena from a location that is not in physical contact with the observed entity, is and has been a tool of discovery and exploration for mankind for quite some time. Primitive man used remote sensing as they stood on a high cliff or treetop in order to search out food or a better place of shelter. Galileo Galilei was a practitioner of remote sensing as he used a telescope to discover three of Jupiter's four moons, Io, Europa and Callisto, in 1610. During World War I, airplanes employed remote sensing as they flew photoreconnaissance missions. [**19**] [**20**] [**21**]

Although the electromagnetic spectrum was first theorized by J.C. Maxwell in 1873, it wasn't until after World War I that remote sensing expanded beyond what the human eye could see, with and/or without the help of magnification. Such a leap was due to advancements in understanding with regard to electromagnetic radiation, spectral reflectance, and the resulting development of various tools that can measure the radiant energy reflected or emitted by objects. It was at this time that remote sensing began to graduate into a multi-disciplinary science involving optics, spectroscopy, photography, computer science, and electronics, amongst others. Since this newfound understanding, spectral imaging accomplishments have continued steadily and now not only facilitate the further exploration of inhabitable or inaccessible areas of our own planet via Earth-orbiting satellites, such as the Landsat, but enable further exploration into deep space via space probes like the Magellan and 2001 Mars Odyssey, as well as countless other civilian and military/security applications. [**19**] [**20**] [**21**]

The rest of this paper will focus on topics related to the usage of spectroscopy in order to utilize the full electromagnetic spectrum for remote sensing purposes.

4.2 On The Electromagnetic Spectrum and Spectral Reflectance

The electromagnetic (EM) spectrum is the name for the many types of radiation (energy) that travel throughout our atmosphere and space. While some of this radiation is visible (violet, blue, green, yellow, orange and red), the majority of radiation is not visible to the naked eye (radio, microwave, infrared, ultraviolet, x-ray and gamma-ray), as Figure 4-1 summarizes. With regard to remote sensing, the majority of scientific and technological advances are now focused on the wavelengths that are not clearly visible – and it is within these wavelengths that spectral reflectance plays an important role.

Spectral reflectance is the ratio of reflected energy (e.g. visible light, microwaves, UV rays, etc) to incident energy for a given wavelength. [**22**] As this function is plotted over a range of wavelengths for a particular material, a curve is formed that can help identify that material. This curve for a particular material is called a spectral signature. [**23**] An example of identifying a material via its spectral signature could be seen in Figure 4-2. Notice how vegetation has a higher spectral reflectance percentage than dry soil in the near infrared

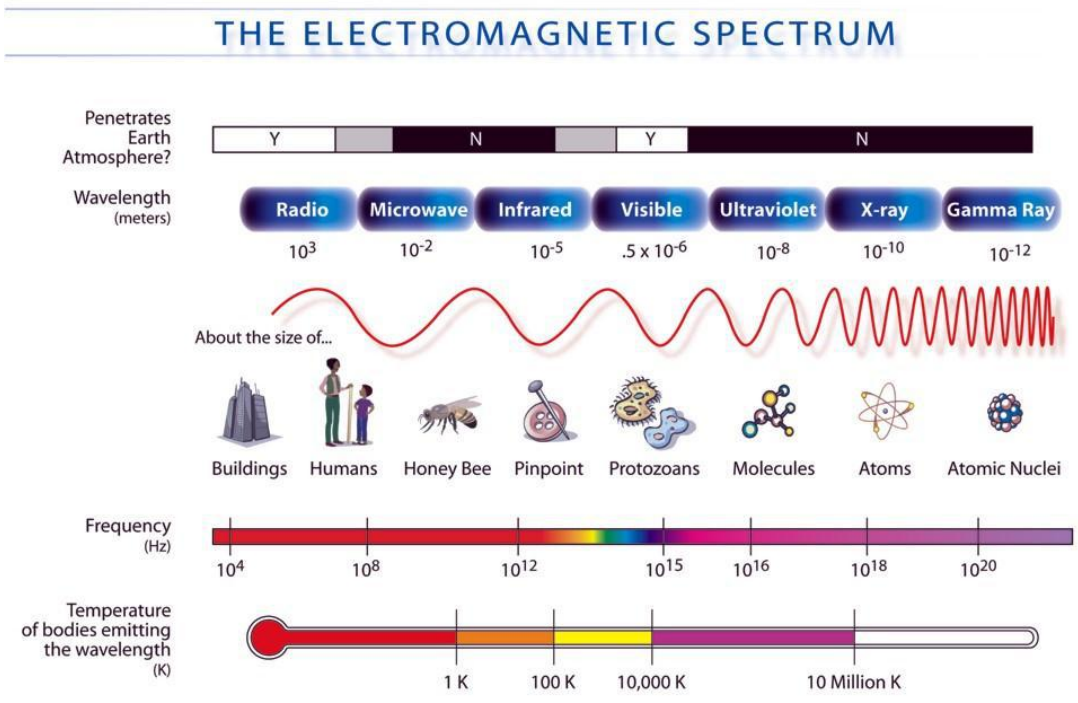


Figure 4-1 - The Electromagnetic Spectrum [15]

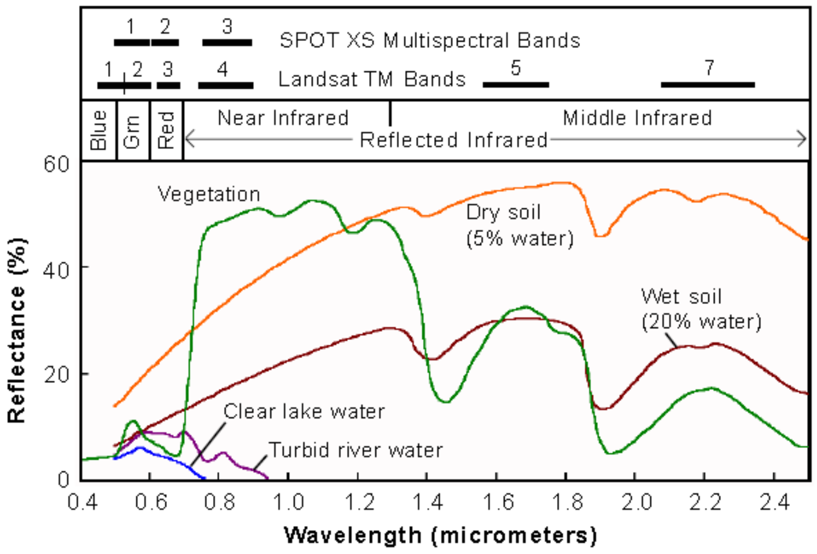


Figure 4-2 - Spectral reflectance curves (signatures) for several common surface materials [19]

range (between .7 and 1.2 micrometers) and a lower reflectance percentage than dry soil in the red range. By remote sensing a scene using equipment that is able to observe reflectance in the near infrared and red wavelengths, one would be able to differentiate vegetation from dry soil – a task that may be quite impossible from a very far distance, or perhaps at night.

4.3 Hyperspectral Imaging

Hyperspectral image processing is a form of remote sensing that has been actively researched and developed over the past decade. [**22**] With this process, information is collected and processed from across the electromagnetic spectrum, in narrow, contagious spectral intervals – much finer in resolution than other types of spectral imaging such as multispectral imagery.

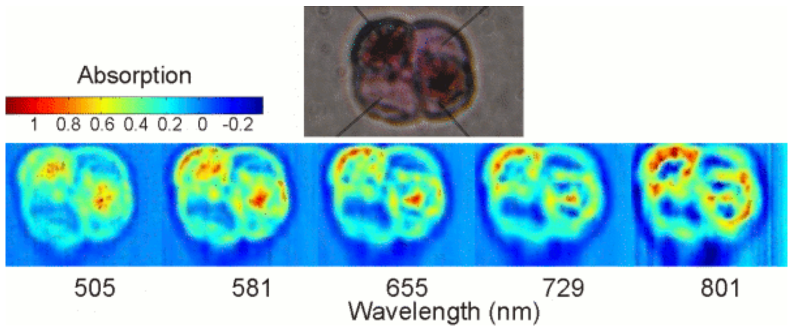


Figure 4-3 - A grain of pollen, along with several images from separate spectral bands [24]

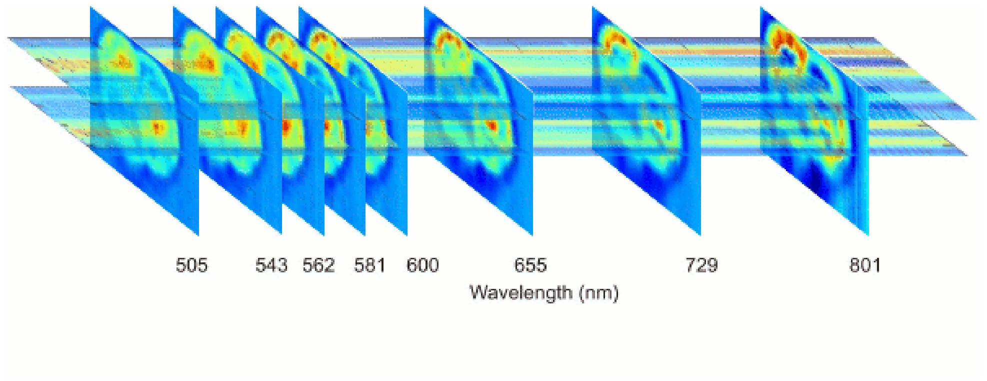


Figure 4-4 - An illustration of a hyperspectral cube obtained from a grain of pollen [24**]**

The more narrow intervals used by hyperspectral imaging provide an enhanced resolution such that relatively small differences in material composition can be observed. Each narrow wavelength sampling is represented by an image (see Figure 4-3). [**24**]

The set of images are then combined to create a three-dimensional "hyperspectral cube" for processing and analysis (see Figure 4-4). Once this cube is created, plotting a single point's relative reflectance across all measures wavelengths would display the spectral curve for the material(s) sampled within that single point, as is shown in Figure 4-5.

Once a spectral curve can be identified, some type of analysis could be made in order to identify the material(s) depicted. One way of doing this is to match an image's spectra (the array of relative reflectance across all measures wavelengths) to the spectral signature of a known material, as may be documented in a published library of spectral signatures. While perfect matches are unlikely, matching reference spectra can be ranked using some agreed upon measure that ultimately would be able to declare one documented signature a "winner" (see Figure 4-6).

The obvious challenge with matching observed spectra with those documented in a reference library is that any exhaustive process with large multi-dimensional data sets as are represented in a hyperspectral cube against a vast library of spectral curves will require an extensive amount of computing power if one wishes to achieve results in a timely manner. [**25**] [**24**] Further, most scenes will include many image pixels that are strongly correlated with their adjacent pixels. If two adjacent materials do not have dissimilar spectra, an edge cannot be easily detected, thereby causing the detection of the material to be more time consuming than is ideal. [**26**] Due to these challenges, amongst others, it would be advantageous to maximize the performance of any image processing algorithms, and minimize the data to be dealt with, prior to any exhaustive measures for material detection. [**27**]

As such, often one would utilize a best band algorithm that is able to find the subset of bands that exhibit maximum seperatability between original and average spectra. Once such bands are obtained, a search can be made on them instead of the full set of spectral measurements as are represented within the hyperspectral cube.

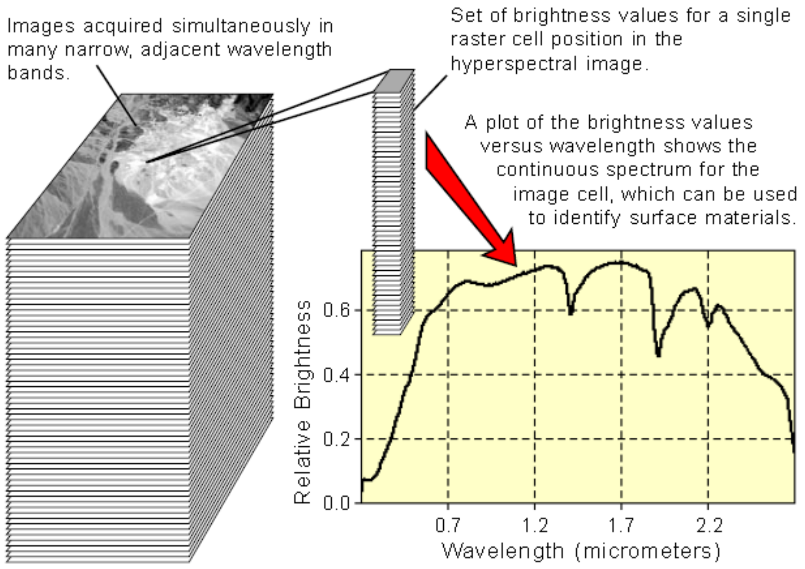

Images acquired simultaneously in many narrow, adjacent wavelength bands.
Set of brightness values for a single raster cell position in the hyperspectral image.
A plot of the brightness values versus wavelength shows the continuous spectrum for the image cell, which can be used to identify surface materials.
Relative Brightness
0.0
0.2
0.4
0.6
0.7
1.2
1.7
2.2
Wavelength (micrometers)

Figure 4-5 - Example of how a single raster cell is represented throughout a hyperspectral cube [22]

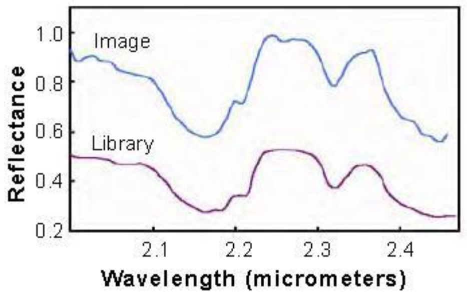


Figure 4-6 - Sample image spectrum and a matched spectrum of the mineral alunite from the USGS Spectral Library (accepted "match" variance of .91mm) [19]

To identify, and maximize, seperatability, one first must design metrics that quantify the distance between two spectra. One such way is to use the spectral angle metric: [**28**] [**26**]

Given two vectors of the same dimension **x** and **y**, the *spectral angle* is defined as the arccosine of their dot product [**28**]:

$$SA(\mathbf{x},\mathbf{y}) = arcos\left(\frac{\langle \mathbf{x},\mathbf{y}\rangle}{\|\mathbf{x}\|\|\mathbf{y}\|}\right) \tag{4-1}$$

where <.,.> represents the dot product of the vectors (a single number obtained by taking two coordinate vectors, multiplying the corresponding entries and adding up the products) and ||.|| represents the Euclidean norm (the absolute value of a complex number). [**29**] [**30**] Then, we can employ methods that select only the bands that represent maximal seperatability:

Given two spectra, **x** and **y** with values over a set of spectra bands **B**, a spectral distance *β*, the goal of band screening is to find the subset of bands $\mathbf{B_1}$ such that [**28**]:

$$\beta(\boldsymbol{x},\boldsymbol{y},B_1) = \min_{B_1 \subseteq B}\left(\beta(\boldsymbol{x},\boldsymbol{y},B)\right) \tag{4-2}$$

where by $\beta(\mathbf{x},\mathbf{y},\mathbf{B_s})$ we refer to the value of the distance measure *β* computed between the two vectors over the subset $\mathbf{B_s}$. [**28**]

Unfortunately, even this process of band screening is computationally complex and demanding. Any hyperspectral image of *n* bands will have 2^n possible mappings to be considered, since given any hyperspectral image of *n* bands, and assuming the B_1 can have any size, the number of band combinations to be tried is roughly equivalent to the number of possible mappings [**26**]:

$$f : \{1,2,3,\dots,n\} \to \{0,1\} \tag{4-3}$$

This is possible since each subset of $\mathbf{B_{s.}}$ and **B** can be seen as an *n*-uple of 0's and 1's where, for each position, one indicating that the corresponding band is in the subset of selected bands and zero indicating the absence of that band. [**26**]

Due to this, it is evident that, even with efficient algorithms, hyperspectral analysis requires the use of powerful computational equipment.

4.4 Case Study on High Performance Computing in Hyperspectral Analysis

4.4.1 Grid Computing for Hyperspectral Data Processing

In May of 2007, a computing grid was deployed at Montclair State University in order to investigate the exhaustive search for best band selection for spectral differentiation. The grid, as depicted in Figure 4-7, was based on Sun N1 Grid Engine Software and employed a variety of PCs throughout the University. At any time, a maximum of 16 PCs were available for computation.

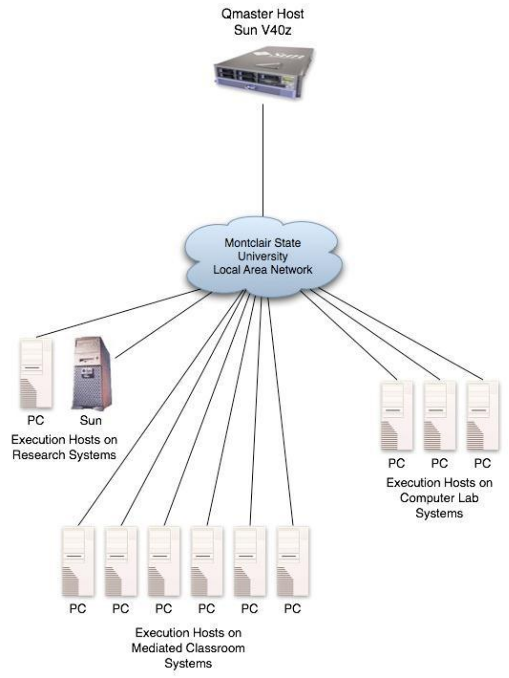

Qmaster Host
Sun V40z
Montclair State University Local Area Network
PC
Sun
Execution Hosts on Research Systems
PC
PC
PC
Execution Hosts on Computer Lab Systems
PC
PC
PC
PC
PC
PC
Execution Hosts on Mediated Classroom Systems

Figure 4-7 - Computing Grid deployed at Montclair State University [26]

10000100000000000000000000000000000000

Figure 4-8 - 38-bit binary sequence used to represent the bands chosen for a given comparison. In this instance, the first and sixth bands have been chosen. [26**]**

In this experiment, with 38-dimensional pixel vectors, the exhaustive search involved 2^{38} mappings (approximately one thousand billion combinations). To minimize the communication time between the grid's master host and the clients, the band combinations were described as binary sequences in which a 1 represents the bands chosen, while a 0 represents a band not chosen for a given comparison [**26**]:

A total of 18 CPUs were available for the test-runs. As would be expected, the execution time was improved proportional to the CPU count, and the overall experiment showed that for such algorithms as exhaustive band search, a distributed approach will tremendously decrease the computational costs.

4.4.2 HPC Cluster for Hyperspectral Data Processing

During the time that the grid was deployed at MSU, no modern computational instrument was available. Although a 64-node cluster was in place, the system was several years old and performance was limited (64GFlops). Under the circumstances, deploying a grid was an excellent approach. While typically not as high-performing as cluster based systems, a grid-based system may provide more consistent run times and is theoretically more scalable since adding more compute nodes only involves running the grid client on a new machine that is connected to the internet/network. Further, the cost is low, since grids utilize existing systems that do not need to be dedicated nodes. [**31**] Lastly, distributed computing systems like a grid do not require special application code to be written, unlike regular parallel computing systems like clusters. [**26**]

However, with the recent grant that Montclair State University received, the University now has access to significantly more computational power (~2.5TFlops). It is with this cluster that we will revisit the exhaustive band search problem.

4.4.2.1 Creating a Parallel Best Band Selection (PBBS) application

Unlike using a grid, in which special code is not required to execute an application in a way that leverages the parallel architecture, the standard tool used by many supercomputers and Beowulf clusters is an implementation of the Message Passing Interface (MPI) communications protocol. [**10**] MPI allows a programmer to implement parallel algorithms that leverage the collective resources of a distributed memory environment. This is accomplished via the calling of MPI routines that enable collective as well as point to point communications between nodes. [**31**]

However, MPI is just a specification. In order to call the set of routines the MPI specification outlines, one must obtain and install the appropriate implementation of MPI for the programming language being used.

With that said, early-on in our work we encountered our first challenge since the original algorithms we wished to parallelize were in Java. While a few MPI implementations for Java are available [**32**] [**33**], currently the most supported and readily available MPI implementations are for Fortran, C and C++. [**10**] [**2**] [**34**]. Further, one of the C/C++/Fortran implementations of MPI, MPICH2, was already installed, tested and working on the cluster. Therefore, it was decided that we would migrate the existing Java code to C/C++ and that we would use the MPICH2 implementation of the MPI specification.

The original code, which can be found in Appendix VIII – Original Java Code, comprised of two separate Java applications and three core functions:

- bestDistance.java
  - input:
    - spectra file
    - starting interval
    - ending interval
  - output:
    - best distance bands
    - minimum distance
  - methods/functions:
    - computeBestDistance: This method contained the algorithm that computes the best distance bands and minimum distance between two intervals.
    - readfileInVectors: This method read in a plain-text file containing the spectra and saved that data into a vector.
- generateQsubDistance.java
  - input:
    - name of spectra file (not ultimately read)
    - size of the vector or number of bands
    - number of jobs, where jobs is defined as 2^{jobs}
  - output:
    - text command(s) that would generate the best distance for the given input. Example:
      - java bestDistance.java <spectra file> <starting interval> <ending interval>
  - methods/functions:
    - generateJobs: This method broke an interval into smaller "jobs" which then would be passed to computeBestDistance.

Each of the above-mentioned methods were ported to C/C++ and combined together into a single application, titled bestDistance.cpp. The design of the new bestDistance.cpp was as follows:

- input:
  - spectra file
  - vector size
  - number of jobs
- output:
  - best distance bands
  - minimum distance
- methods/functions:
  - computeBestDistance: This method contained the algorithm that computes the best distance bands and minimum distance between two intervals.
  - readfileInVectors: This method read in a plain-text file containing the spectra and saved that data into a vector.
  - generateJobs: This method broke an interval into smaller "jobs" which then would be passed to computeBestDistance.

Once the code was ported to C/C++, the next step was to determine how to convert the serial application into one that leverages a parallel environment via MPI.

Ultimately, it was decided that the computeBestDistance function performed the most work and that a significant improvement in Application Performance as well as Application Run Time could be easily achieved if we spread that work across the nodes. Yet, to do so, we had to determine the method of spreading the work. The first approach of spreading the work, and what we ultimately determined to be the best approach, was to have a master node distribute a pre-determined number of calls to computeBestDistance to all nodes, and then distribute additional calls until all calls are completed. However, implementing this approach seemed at first to be too time-consuming for the scope of this project. The second approach to distributing the work of computeBestDistance was to divide the total number of calls to computeBestDistance up-front, and then distribute all of the work immediately, as opposed to "as-needed". This second approach also had its challenges, but these were more easily addressed in the time frame we had available. As such, we went with the second approach.

The first challenge was to determine how to distribute the total number of calls to all compute nodes. Since the generateJobs function computed the total calls to computeBestDistance as a factor of the run-time parameter for number of jobs (J), where:

$$actual\ jobs = 2^J - 1 \tag{4.1}$$

there would not always be an even number of calls to computeBestDistance. Example: a run-time "number of jobs" input of 10 would result in 1,023 jobs. If 64 nodes were being used, you could not evenly divide 1,023 calls to computeBestDistance by 64 nodes. The solution for this problem was to spread the closest evenly divisible number of calls to computeBestDistance amongst all compute nodes, and then have the master compute the difference. The pseudo-code to accomplish this was as follows:

```
if (total number of jobs is evenly divisible by the total number of nodes)
        jobs per slave = total jobs / total nodes;
        jobs per master = jobs per slave;

else if (total number of jobs is NOT evenly divisible by the total number of nodes)
        jobs per slave = (total number of jobs – (total number of jobs % total number of nodes) / total number of nodes);
        jobs per master = jobs per slave + (total number of jobs % total number of nodes);
```

As was made evident during the experimental runs, this method was not the most ideal method. Yet, for sake of time, we were not able to make the necessary changes in the code. For sure, future use of this Parallel Best Band Selection algorithm should address this coding flaw (see Future Work).

Once the decision was made on which functions to parallelize and how to divide up the work, we began to implement the necessary MPI calls. It was during this time that we encountered our next challenge. Although our design called for the reading of the spectra file to only be done by the master node, and then for the data to be sent to all other nodes, it turned out that this would be easier said than done. We found that, due to the primitive data types that MPI requires for its various send, receive and broadcast functions, the sending of a vector simply cannot easily be done directly. For sake of time, we chose to instruct all nodes to read in the spectra file, since it is a fairly quick task and does not dramatically impact Application Run Time and Application Performance.

The final design of the application is as follows:

- All Nodes:
    - MPI initialization
    - Variable declaration
    - Read in the spectra input file

    - Receive assigned jobs
    - Execute computeBestDistance for each assigned job
    - Send the final results to the Master Node
- Master Node Only
    - Read in the run-time input arguments; ensure that syntax is correct
    - Generate the jobs
    - Determine the number of jobs to send to each node, and the number of jobs that the master must perform
    - Distribute the jobs
    - Start wall clock timer
    - Compute assigned jobs
    - Receive completed jobs
    - Stop wall clock timer
    - Print the results

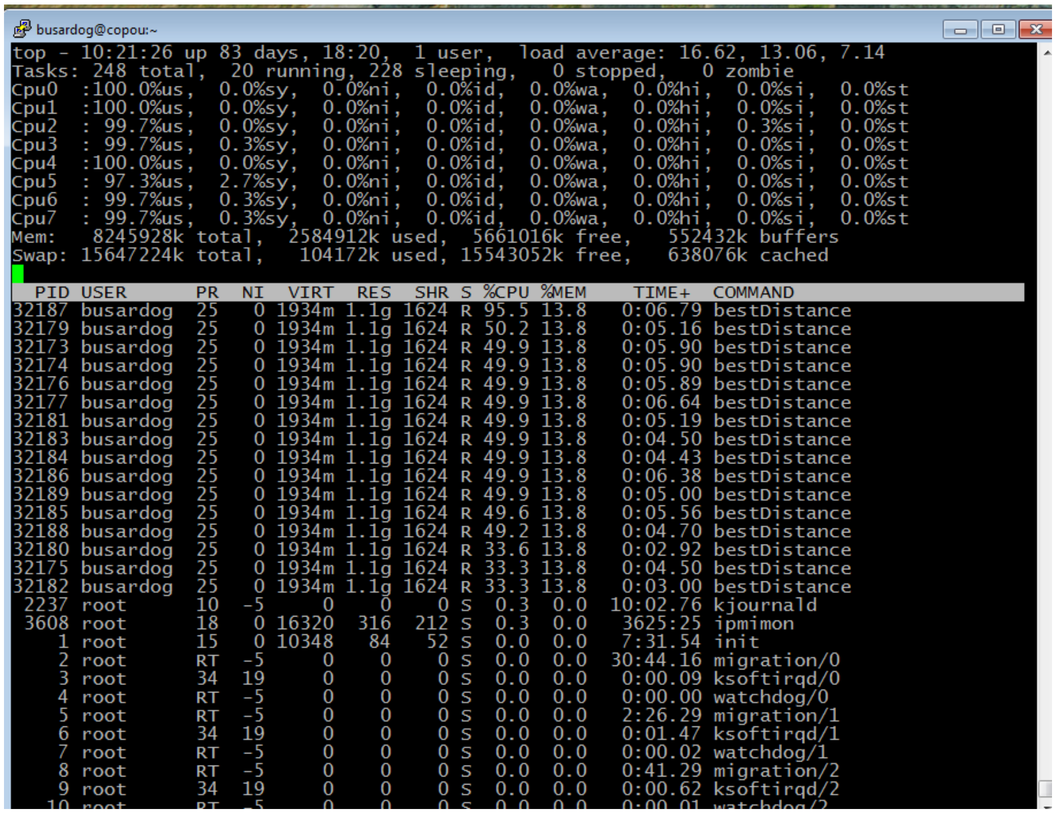
Details regarding how MPI was used to allow for the above-mentioned design can be found in Appendix III – Leveraging MPI for Parallelization of C/C++ Code.

After parallelization via MPI was completed, we decided to also implement threading capabilities on the computeBestDistance computations. By doing so, we would be able to utilize all (8) cores of each node on the cluster. First, we allowed for an additional run-time parameter so that a user can enter the number of concurrent threads to execute on each node. Then, we made the following adjustments in how computeBestDistance was called:

- For each node, computeBestDistance was called via a new thread, up to the total number of concurrent threads defined
- Since each call to computeBestDistance completes at approximately the same time, the code would wait for all threads to complete before collecting results and repeating the process, up to the total number of assigned jobs

An example of the improved efficiency achieved by the implementation of threading can be seen in Figure 4-9. For additional details regarding how threading was implemented, see Appendix IV – Leveraging "pthreads" for Symmetric Multiprocessing (SMP).

A final adjustment to the code was with its output. After several very large jobs were executed (e.g. jobs that resulted in over 500,000 calls sent to computeBestDistance), it was observed that Application Run Time was negatively affected via the overhead required to print all of the intervals to be sent to computeBestDistance, as well as to save and print all of the results. While these functions would be necessary should the actual results be required, such functions would ideally be disabled if only Application Performance was to be measured. As such, an additional run-time parameter was allowed in order to request a "performance mode" that disabled the saving and displaying of any output aside from performance data such as start and stop time. This special "performance mode" could be requested at run-time via a trailing *–performance* argument.



```
top - 10:21:26 up 83 days, 18:20,  1 user,  load average: 16.62, 13.06, 7.14
Tasks: 248 total,  20 running, 228 sleeping,   0 stopped,   0 zombie
Cpu0  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  : 99.7%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu3  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu4  :100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu5  : 97.3%us,  2.7%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  : 99.7%us,  0.3%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8245928k total,  2584912k used,  5661016k free,   552432k buffers
Swap: 15647224k total,   104172k used, 15543052k free,   638076k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
32187 busardog  25   0 1934m 1.1g 1624 R 95.5 13.8   0:06.79 bestDistance
32179 busardog  25   0 1934m 1.1g 1624 R 50.2 13.8   0:05.16 bestDistance
32173 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:05.90 bestDistance
32174 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:05.90 bestDistance
32176 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:05.89 bestDistance
32177 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:06.64 bestDistance
32181 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:05.19 bestDistance
32183 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:04.50 bestDistance
32184 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:04.43 bestDistance
32186 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:06.38 bestDistance
32189 busardog  25   0 1934m 1.1g 1624 R 49.9 13.8   0:05.00 bestDistance
32185 busardog  25   0 1934m 1.1g 1624 R 49.6 13.8   0:05.56 bestDistance
32188 busardog  25   0 1934m 1.1g 1624 R 49.2 13.8   0:04.70 bestDistance
32180 busardog  25   0 1934m 1.1g 1624 R 33.6 13.8   0:02.92 bestDistance
32175 busardog  25   0 1934m 1.1g 1624 R 33.3 13.8   0:04.50 bestDistance
32182 busardog  25   0 1934m 1.1g 1624 R 33.3 13.8   0:03.00 bestDistance
 2237 root      10  -5     0    0    0 S  0.3  0.0  10:02.76 kjournald
 3608 root      18   0 16320  316  212 S  0.3  0.0   3625:25 ipmimon
    1 root      15   0 10348   84   52 S  0.0  0.0   7:31.54 init
    2 root      RT  -5     0    0    0 S  0.0  0.0  30:44.16 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.09 ksoftirqd/0
    4 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   2:26.29 migration/1
    6 root      34  19     0    0    0 S  0.0  0.0   0:01.47 ksoftirqd/1
    7 root      RT  -5     0    0    0 S  0.0  0.0   0:00.02 watchdog/1
    8 root      RT  -5     0    0    0 S  0.0  0.0   0:41.29 migration/2
    9 root      34  19     0    0    0 S  0.0  0.0   0:00.62 ksoftirqd/2
   10 root      RT  -5     0    0    0 S  0.0  0.0   0:00.01 watchdog/2
```

Figure 4-9 - Example CPU Utilization for 16 Concurrent Threads

The full C/C++ code along with output examples can be found in Appendix II - Application Source Code.

4.4.2.2 Experimental Runs

Four experiments were performed in order to test whether the PBBS algorithm within an HPC environment provides a significant improvement over traditional sequential methods, as well as to examine the behavior of the algorithm for different inputs. The data set used for these experiments was a Hyperspectral Digital Imagery Collection Experiment (HYDICE) image corresponding to part of the Forest Radiance set [**35**]. The data are 16 bit reflectance values organized in 210 bands spanning the 400 to 2500nm range and collected

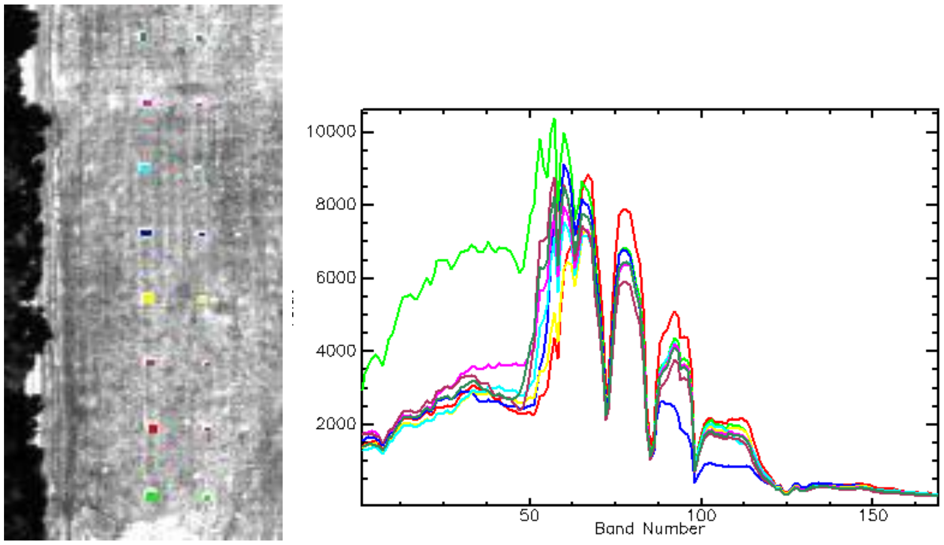


Figure 4-10 - a) Hydice hyperspectral data displaying rows of panels marked by squares, b) Average spectra for the eight panel categories in the scene

(a) (b)

with a spatial resolution of 1.5m. The Forest Radiance data is provided by the Spectral Information Technology Application Center (SITAC), and is often found in published research involving targets. Figure 4-10a provides a view of a sub scene of the large data with points of interest highlighted. The area presents a special advantage through 24 man-made panels placed in 8 rows on the ground in which each of the 3 columns contains a different material, while each row contains panels of different sizes (3m by 3m, 2m by 2m, and 1m by 1m respectively). The sizes mean that the third row panels are basically smaller than the spatial resolution, and thus, the pixels covering them will have to be inherently mixed. The average spectra for each of the materials are plotted in Figure 4-10b. For the purpose of our experiment we focused on the first row of panels, our focus being identification of the best bands subset that will minimize the dissimilarity among the spectra for the corresponding material. Four spectra were manually selected from the panels and used as start for our PBBS algorithm.

Experiment 1: Shared memory single node performance. The purpose of the first experiment was to observe and compare the computational performance of a single-node (non-MPI), on a single core versus a multi-threaded approach utilizing all (8) of the node's cores. The vector size of (n = 34) bits was chosen since it could complete a serial computation within a reasonable amount of time. The sequential run completed the best band computation in 612.662 minutes. To understand the impact of the number of intervals k we have varied it from 1 to 1023. Figure 4-11 provides the speedup obtained as the number of intervals increases. The speedup was computed as the ratio between the execution time for *k-1* and the execution time for the current *k.* As expected, as k increases, the performance decreases since division in smaller intervals brings only overhead to the execution time. We note however that, even for large k, the overhead is limited to only 50% of the execution time.

We also analyzed the robustness of our algorithm on a parallel shared memory environment by executing a multithreaded version on a single machine and varying the number of threads from 1 to 16. Figure 4-12 provides the speedup computed as the ratio between the single thread execution time and the multiple thread execution times. In all cases k was the same (1023). To help with understanding the results, an ideal speedup is plotted with a dashed line. The algorithm performs well for 8 threads (speedup 7.1) and records only minimal improvement for 16 (speedup 7.73). This is explained by the configuration of our nodes, which have only 8 computing cores.

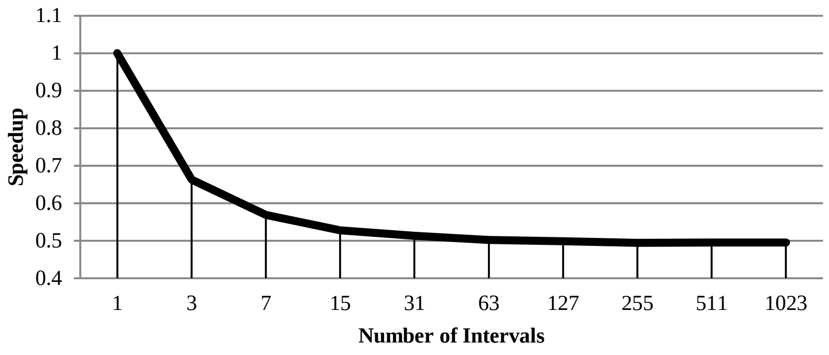


Figure 4-11 - Sequential execution of the Best Band Selection algorithm for n=34 and k varied from 1 to 1023.

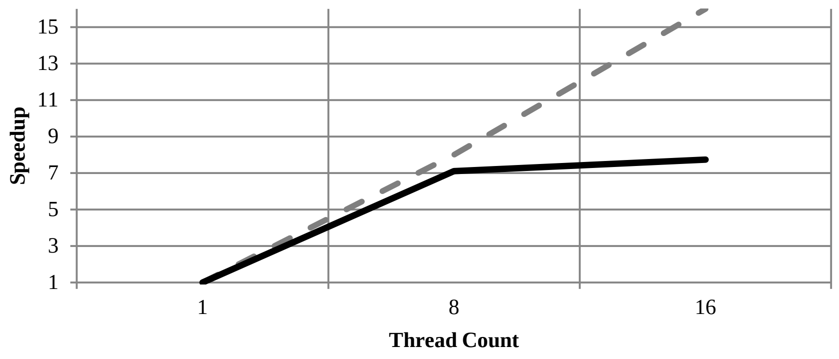


Figure 4-12 - Shared memory multithreaded execution of PBBS for varying number of threads and k set to 1023.

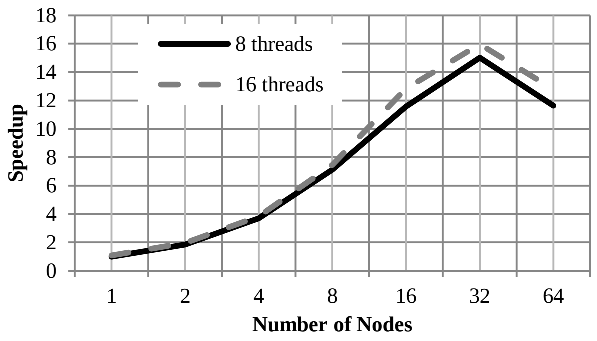


Figure 4-13 - PBBS performance as the number of cluster nodes used increases.

Experiment 2: Beowulf cluster performance. The second experiment was an expansion on the first, via the introduction of parallelization using MPI. The goal of this experiment was to confirm that significant performance improvements would be achieved with further distribution of workload. To do so, various parallel runs of the same, (n = 34) vector size and k = 1,023 intervals were performed on an increasing number of nodes. Both 8 and 16 threads per node were used.

With 2 nodes and 16 simultaneous threads per node, the application completed in 43.8968 minutes. Figure 4-13 provides a summary of the runs expressed as the speedup over the 8 thread single node execution. The solid line corresponds to the 8 thread/node execution while the dashed line corresponds to the 16 thread/ node execution. We see that the speedup for both 8 and 16 threads is similar. In both cases, as the number of nodes increases beyond 32 the performance decreases. In analyzing such behavior

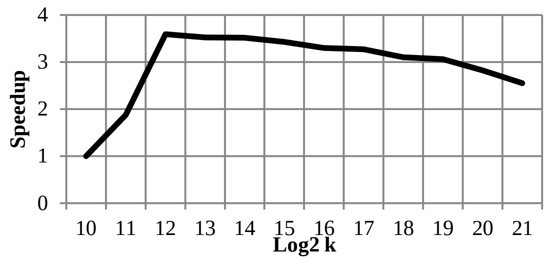


Figure 4-14 - PBBS performance as the number of jobs increases

we noted that in our implementation the master node is also receiving execution jobs and becomes an execution bottleneck. Moreover, as the number of nodes increases, the number of intervals allocated for each node is no longer balanced, resulting in one or more nodes having extended execution times. A reanalysis of the code and a better job balancing is expected to improve the results (see Future Work).

Experiment 3: Impact of k. The third experiment focused on full-cluster runs of increasing interval sizes (i.e. decreasing *k*), with a goal of observing any improvements in average job completion times. The vector size of (n = 34) remained the same, as did the use of 16 concurrent threads.

This experiment started with k = 2,047 intervals, which provided an average time per job of 0.078697 seconds. When the total intervals increased to 4,095, the average time per job dropped to .0206025 seconds. Figure 4-14 provides a summary of the speedups computed as the ratio between the total execution time for k=2^{10} and the total execution time as k varies between 2^{10} to 2^{21} (i.e. 2,097,151 intervals). We see that, beyond a significant increase up to 2^{12} the total execution time is no longer increased or decreased. This behavior can be explained by the fact that as the interval sizes decrease the overhead introduced by the communication increases.

Experiment 4: Robustness estimation. The fourth experiment was a further expansion on each of the previous experiments, but with larger vector sizes. In this experiment, n = 38 spectra were used and the computation performed for best distance using a single core and one interval (k=1). This first run required 5,326.2 minutes to complete its one job. This same vector size was then computed again on a single node, but split into 1,023 intervals which were distributed over the node's (8) cores via threading. This second run completed in 1,384.78 minutes (1.3536 minutes per job, on average). The third run of this experiment distributed the same 1,023 intervals across the full cluster (64 compute nodes + 1 master node) via MPI, which completed in 83.5635 minutes (0.08168 minutes per job, on average). Figure 4-15 provides a visual summary of these three runs.

In Figure 4-16, we show a summary of the execution times for n=38 and k = 10, 20, 21 and 22 respectively. We note that again, as the number of intervals increases beyond 2^{20} no performance improvement is observed.

Finally, we have also investigated the robustness of the algorithm as the vector size continues to increase. Table 4-1 provides a summary of these experiments. The size of the spectra used varied from 34 to 44 while the number of intervals doubled at each increase. Problem size refers to how much larger the problem becomes as the vectors increase in size. The execution time provided is given in minutes. The last column (*Ratio*) refers to the ratio between the execution time for n = 34 to the execution time of the current experiment. The results show that as n increases the execution time remains proportional to 2^n. Based on such experiments one can predict the execution time for larger vector sizes. Given that for n=44 the application completes in more than 15 hours it is clear that significantly larger clusters must be used for a vector size beyond 50 or so dimensions.

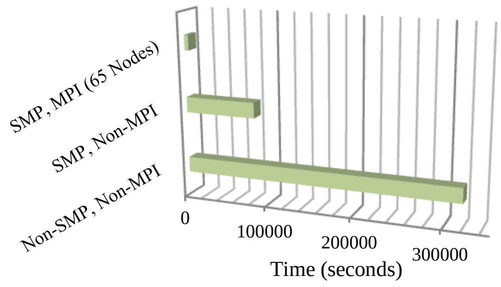


Figure 4-15 - PBBS performance for n=38 and three different approaches: full cluster, single node multithreaded, and single node sequential.

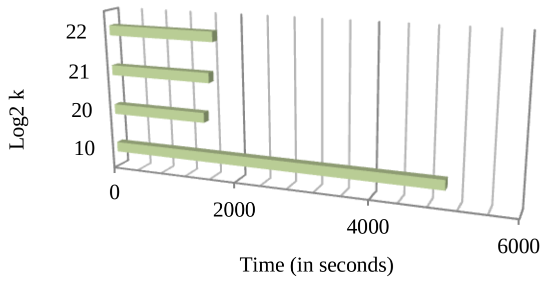


Figure 4-16 - PBBS performance as the number of jobs increases.

nn	*Problem Size*	*kk*	*Execution time*	*Ratio*
334	**1**	19	1.64796	**1**
338	**16**	20	24.8205	**15.06135**
442	**256**	21	400.355	**242.9398**
444	**1024**	22	1643.01	**996.9963**

Table 4-1 - PBBS Performance Analysis

5 Conclusions

This project allowed for an in-depth analysis on Beowulf clusters, parallel computing, and the methods of testing the capabilities of such tools. The cluster's first scientific project, the Parallel Best Band Selection (PBBS) algorithm, not only allowed us to begin learning how to use the new instrument now available to staff and students at MSU, but the experimental results of the PBBS algorithm showed that parallel implementations when run on a modern, distributed-memory Beowulf cluster work very well for exhaustive best band selection, a key method in hyperspectral imagery. Finally, much of the research, code and lessons-learned while working on this new equipment have the potential to form a foundation and spring-board for future related work on the equipment.

6 Future Work

Both the benchmarking performed, as well as the PBBS application, barely scratched the surface in testing the full capabilities of the new cluster at MSU. Future expansion on the Benchmarking work performed within this project could comprise of:

- Additional tuning of the HPL benchmark in order to achieve a higher Percentage of Peak
- Baseline benchmarking, which could be kept on file and leveraged in the future should there be reason to compare current performance to any baseline (e.g. post hardware or software upgrade)
- Download, compilation and installation of the latest version of the NPB Suite in order to see how the increased problem size within the "E" class impacts cluster performance

With regard to the PBBS application, future expansion on the progress made within this project could include:

- General review and optimization of the code
- Implement a more efficient method of dividing the workload across the nodes
    - Suggestion is to have the master distribute a user-defined number of concurrent calls to computeBestDistance to each node that is available until all available nodes received their "batch". Then, as each node completes their assigned jobs, a signal is sent back to the master in order to send additional work if additional work is available.
- Adjusting the code so that the master node is no longer a computation node, by default. Instead, the master can handle the following:
    - Read in the spectra file
    - Generate the jobs
    - Distribute the jobs to the nodes
    - Collect and print out the results

References

[1] Wikipedia. (2010) Beowulf (computing). [Online]. http://en.wikipedia.org/wiki/Beowulf_(computing)

[2] Argonne National Laboratory. MPICH2 : High-performance and Widely Portalable MPI. [Online]. http://www.mcs.anl.gov/research/projects/mpich2/

[3] David H Bailey, "How Fast Is My Beowulf," *Beowulf Cluster Computing with Linux*, pp. 154-157, 2002.

[4] Sandia National Labs. Cbench - Scalable Cluster Benchmarking and Testing. [Online]. http://sourceforge.net/apps/trac/cbench/#DocumentationandResources

[5] Russell Coker. Bonnie++ now at 1.03e. [Online]. http://www.coker.com.au/bonnie++/

[6] IOzone.org. IOzone Filesystem Benchmark. [Online]. http://www.iozone.org/

[7] Compute Canada. DGEMM Benchmark. [Online]. https://computecanada.org/?pageId=138

[8] University of Virginia. Stream Benchmark Reference Information. [Online]. http://www.cs.virginia.edu/stream/ref.html

[9] Rick Jones. Welcome to Netperf. [Online]. http://www.netperf.org/netperf/NetperfPage.html

[10] Wikipedia. Message Passing Interface. [Online]. http://en.wikipedia.org/wiki/Message_Passing_Interface

[11] Netlib.org. Linpack Benchmark -- Java Version. [Online]. http://www.netlib.org/benchmark/linpackjava/

[12] Netlib. (2008, September) HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Systems. [Online]. http://www.netlib.org/benchmark/hpl/

[13] Innovative Computing Laboratory at the University of Tennesee. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. [Online]. http://www.netlib.org/benchmark/hpl/

[14] Netlib. HPL Tuning. [Online]. http://www.netlib.org/benchmark/hpl/tuning.html

[15] National Aeronautics and Space Administration. NAS Parallel Benchmarks. [Online]. http://www.nas.nasa.gov/Resources/Software/npb.html

[16] Wikipedia. (2010) List of device bit rates. [Online]. http://en.wikipedia.org/wiki/List_of_device_bit_rates

[17] Lawrence Livermore National Laboratory. (2010, Mar.) Message Passing Interface (MPI). [Online]. https://computing.llnl.gov/tutorials/mpi/

[18] National Energy Research Scientific Computing Center (NERSC). (2010 , Jan.) NERSC MPI Tutorial. [Online]. http://www.nersc.gov/nusers/help/tutorials/mpi/intro/

[19] Wikipedia. (2010) Remote sensing. [Online]. http://en.wikipedia.org/wiki/Remote_Sensing

[20] Wikipedia. (2010) Galileo Galilei. [Online]. http://en.wikipedia.org/wiki/Galileo_Galilei

[21] Shefali Aggarwal, "Principles of Remote Sensing," *Satellite Remote Sensing and GIS Applications in Agricultural Meteorology*, no. Proceedings of the Training Workshop , pp. 23-38, July 2003.

[22] Ph.D Randall B. Smith. (2006) MicroImages Tutorials: Learning Geospatial Analysis. [Online]. http://www.microimages.com/getstart/pdf/hyprspec.pdf

[23] Wikipedia. (2010) Spectral signature. [Online]. http://en.wikipedia.org/wiki/Spectral_signature

[24] Wikipedia. (2010) Hyperspectral imaging. [Online]. http://en.wikipedia.org/wiki/Hyperspectral_imaging

[25] Richard B. Gomez and Thomas Boggs, "Fast hyperspectral data processing methods," School of Computational Sciences, George Mason University, Fairfax, VA,.

[26] Nicholas A. Senedzuk Stefan A. Robila, "Grid computing for hyperspectral data processing," Center for Imaging and Optics, Dept of Computer Science, Montclair State University, Montclair, NJ, 2006.

[27] Wikipedia. (2010) Feature extraction. [Online]. http://en.wikipedia.org/wiki/Feature_extraction

[28] F. A. Kruse et al., "The Spectral Image Processing System (SIPS) - interactive visualization and analysis of imaging spectometer data," *Remote Sensing of the Environment*, no. 44, pp. 145-163, 1993.

[29] Wikipedia. (2010) Norm (mathematics). [Online]. http://en.wikipedia.org/wiki/Euclidean_norm#Euclidean_norm

[30] Wikipedia. (2010) Dot product. [Online]. http://en.wikipedia.org/wiki/Dot_product

[31] Jason Brazile et al., "Cluster versus grid for large-volume hyperspectral image preprocessing," Remote Sensing Laboratories, Department of Geography, University of Zurich, Zurich, Switzerland, Paper 20040804, 2004.

[32] Pervasive Technology Labs at Indiana University. (2007, May) The HPJava Project. [Online]. http://www.hpjava.org/mpiJava.html

[33] University of Reading. MPJ Express Project. [Online]. http://mpj-express.org/

[34] Then Open MPI Project. Open MPI: Open Source High Performance Computing. [Online]. http://www.open-mpi.org/

[35] S. Bergman, R. C. Olsen, and R. G. Resmini, "Target detection in a forest environment using spectral imagery," vol. Imaging Spectrometry III, no. Imaging Spectrometry III, pp. 46-56, October 1997.

[36] Argonne National Laboratory. Frequently Asked Questions - MPICH2. [Online]. http://wiki.mcs.anl.gov/mpich2/index.php/Frequently_Asked_Questions

[37] Automatically Tuned Linear Algebra Software (ATLAS). [Online]. http://math-atlas.sourceforge.net/

[38] Microway. Microway. [Online]. http://www.microway.com/mmds.html

[39] Microway. Microway - Technology you can count on, since 1982. [Online]. http://www.microway.com

[40] Trustees of Indiana University. LAM/MPI Parallel Computing. [Online]. http://www.lam-mpi.org/

[41] Montclair State University. Stefan Robila, Computer Science, $190,010 award from NSF. [Online]. http://csam.montclair.edu/csam_news/article.php?ArticleID=4103&ChannelID=54

[42] Innovative Computing Laboratory at the University of Tennesee. HPC Challenge Benchmark. [Online]. http://icl.cs.utk.edu/hpcc/

[43] Chris Armstrong, "Ensuring Value in HPC Procurements - A few benchmarking tips from HECToR," *Scientific Computing*, 2009. [Online]. http://www.scientificcomputing.com/articles-HPC-Ensuring-Value-in-HPC-Procurements-112009.aspx

[44] Argonne National Laboratory. The Message Passing Interface (MPI) standard. [Online]. http://www.mcs.anl.gov/research/projects/mpi/

[45] Top500.ORG. Top500 Supercomputing Sites. [Online]. http://www.top500.org/lists

[46] Wikipedia. FLOPS - Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/wiki/FLOPS

[47] Lawrence Livermore National Laboratory. Introduction to Parallel Computing. [Online]. https://computing.llnl.gov/tutorials/parallel_comp/#Designing

[48] Sandia National Labs. doc/HOWTO-RunTestHowTo_1.1. [Online]. http://sourceforge.net/apps/trac/cbench/wiki/doc/HOWTO-RunTestHowTo_1.1

[49] Wikipedia. General Matrix Multiply. [Online]. http://en.wikipedia.org/wiki/General_Matrix_Multiply

[50] Intel. Intel MPI Benchmarks 3.2. [Online]. http://software.intel.com/en-us/articles/intel-mpi-benchmarks/

[51] GeoMart. (2010) CIR (Color Infrared) Aerial Photography - Explained. [Online]. http://www.geomart.com/products/aerial/cir.htm

[52] NASA. (2007, July) The Electromagnetic Spectrum. [Online]. http://mynasadata.larc.nasa.gov/images/EM_Spectrum3-new.jpg

[53] P&P Optica. (2010) Hyperspectra Spectroscopy. [Online]. http://www.ppo.ca/Hyperspectral_imaging.htm

[54] Thomas Boggs and Richard B. Gomez, "Fast hyperspectral data processing methods," School of Computational Sciences, George Mason University, Fairfax, VA,.

[55] National Energy Research Scientific Computing Center (NERSC). (2010) NERSC MPI Tutorial. [Online]. http://www.nersc.gov/nusers/help/tutorials/mpi/intro/basic.php

Appendix I - RFQ Summary

The following was written and provided to several vendors prior to the purchase from Microway:

Montclair State University requires a Linux-based High Performance Computing Cluster with 512 x86-64bit computer cores and switched Gigabit Ethernet connectivity. This system should have at least 1TB of RAM, 5TB of storage, and should be able to achieve at least 2.5TFlops (validated via Linpack Benchmark).

The processors and equipment used in this cluster should be as energy-efficient and reliable as they are high-performing. The cluster must come with one to two cabinets that, when combined with the management console, take up no more than 12' by 10' of space within one of the University's state of the art computing facilities. The facilities will be prepared for a maximum power requirement of 20kw and a cooling requirement of 89,000BTU. The cabinets should be designed to maximize ventilation and cooling.

In addition to the cluster, a master node and two dedicated workstations will be needed that have a similar OS, CPU and memory configuration as the individual cluster nodes in order to allow cluster administration (via master node) and initial testing of applications before full deployment on the cluster (via workstations).

The Operating System on each cluster node as well as workstation should be Red Hat Linux Enterprise (5), of which the University will require media and a 3-5 year license. The parallel communication software should be based on MPICH2 libraries, and the GNU compiler should be pre-installed for C and C++ cluster programming. Some type of cluster management software should also be provided so that an administrator can control the cluster from the master node or any remote location in order to administer the unit (e.g. backup/reimage individual nodes or restart/shutdown individual nodes or the entire cluster).

The final product should be pre-configured and fully tested prior to delivery, have a strong off-site warranty and (preferably) come with lifetime technical support via phone, fax and email.

RFQ Terms and Conditions

- Any additional costs must be detailed in your quotation.
- Labor and shipping costs are required.
- Technical inquires of any nature can be addressed to Stefan Robila (robilas@mail.montclair.edu).
- Email completed quotations to Stefan Robila (robilas@mail.montclair.edu).

Technical Specifications

What follows is a proposed configuration based on the needs as stated within the above Summary section. It includes a master node and 64 compute nodes consolidated into 32 Twin Dual Opteron chassis with each compute node having 8 processor cores and 16GB memory. Also included are two workstation configurations, rackmount cabinets, network switches, a LCD, Red Hat Linux server and workstation license and media as well as cluster management software.

(1) Quad-Core Opteron Master Node

- 1U Dual Opteron Server chassis with/supporting
  - Two Quad-Core / Dual-Core AMD Opteron 2000-series Processors
  - Up to 64GB DDR2 800/667/533 Memory (8 Slots)
  - Left Universal Slot:
    - 1 64-bit PCI-X 133MHz OR (x8) PCI-E
  - Right Universal Slot:

        - 1 64-bit PCI-X 133MHz OR (x8) PCI-E
    - MCP55 Pro Chipset Dual-Port Gigabit Ethernet Controller
    - 4 x 3.5" Hot-swap SATA Drive Bays
    - Slim-line CD/DVD and Floppy Drive Bays
    - Rackmount Rail Kit
- (2) AMD "Shanghai" Quad Core Socket F Opteron 2378
    - 2.4 GHz with 4x 512 KB Cache/core and 6 MB Shared
- (4) 2GB DDR2 800 MHz ECC/Registered Memory (Total of 8GB)
- (2) 500 GB Seagate Barracuda ES.2 SATA/300 ST3500320NS
    - 32MB Cache, 3Gb/s, NCQ, 7200RPM, 1.2 million hours MTBF
    - Maximum Sustained Transfer Rate: 105 MB/sec
    - (Linux Software-based) RAID1 Mirror
- (2) Cat 6 Ethernet Cable (10 ft)
- Red Hat Enterprise Linux 5 Server for HPC Compute Nodes Basic (x86, AMD64, EM64T)
    - Supports up to 2 CPU Sockets
    - 3 Year Subscription with E-mail Support
- Red Hat Enterprise Linux 5 Server (Media)
- Red Hat Enterprise Linux 5, MCMS, MPICH2, Torque, Maui Scheduler & GNU Compilers installed and configured.

(32) Quad-Core Opteron Twin-Compute Node (Total of 64 Compute Nodes)

- 1U Twin Dual Opteron Chassis with/supporting
    - Redundant Chassis Cooling Fans
    - Rackmount Rail Kit
- Each of the Twins to include:
    - Two Dual-, Quad- or Six-Core Opteron Processors
    - NVIDIA MCP55 Pro chipset with HyperTransport 3.0 Link Support
    - Up to 64GB DDR2-800/667/533 Memory (16 DIMMs)
    - 2x Hot-swap SATA-II Drive Bays
    - 2x Dual Integrated NVIDIA MCP55V-Pro Gigabit Ethernet Ports
    - 1x Integrated XGI Z9S 32MB PCI Graphics
    - 1x PCI-E x16 Low-Profile Expansion Slot
    - 2x USB 2.0 Ports, External Serial Port
- (4) AMD "Shanghai" Quad Core Socket F Opteron 2378
    - 2.4 GHz with 4x 512 KB Cache/core and 6 MB Shared
- (8) 4GB DDR2 800 MHz ECC/Registered Memory (16GB Memory per Node)
- (2) 80 GB Seagate SATA/300 (8MB/7200rpm/NCQ) ST380815AS
- (2) Cat 6 Ethernet Cable (10 ft)
- (2) Red Hat Enterprise Linux 5 Server for HPC Compute Nodes Basic (x86, AMD64, EM64T)
    - Supports up to 2 CPU Sockets
    - 3 Year Subscription with E-mail Support
- Red Hat Enterprise Linux 5, MCMS, MPICH2, Torque, Maui Scheduler & GNU Compilers installed and configured.

(1) 96-Port Expandable Gigabit Ethernet Fabric

- (2) Netgear 48 Port (96Gbps + 20Gbps Stacking) 10/100/1000 Smart Switch
    - Includes Two Rear Ports for Switch Stacking (up to 6 switches - 288 Ports)

(1) GIS 1U 15" Rackmount Monitor with USB Keyboard and Touchpad

(2) 42U Rackmount Cabinets with Power Distribution

- APC NetShelter SX 42U Enclosure
    - Standard 19" Rackmount Cabinet (up to 36" mounting depth)
    - Includes casters, perforated doors and solid side panels,
    - Casters and leveling feet,
    - Locks on front and rear door,
    - Adjustable mounting channels,
    - Dimensions: 79" H x 24" W x 42" D
    - Dimensions (packaged): 84" H x 36" W x 48" D
    - (http://www.apcc.com/resource/include/techspec_index.cfm?base_sku=AR3100)
- (4) APC AP7932 Zero U, Switched Rack PDU (120V 2.9kW)
    - Input (120V): L5-30P on 10 Foot Cord
    - Output: 24x NEMA 5-20R
    - Provides remote load monitoring, as well as per-outlet on/off capability.
    - (http://www.apc.com/resource/include/techspec_index.cfm?base_sku=AP7932)

Cluster Management Software

- Includes a monitoring and control software package that integrates the tools needed to understand what events are taking place on the cluster and the tools to control the cluster itself
    - Provides the capability of monitoring the cluster from any location via a web-based interface.
    - Provides the capability of controlling the cluster from the master node
    - Backup and Restore Node System Images
    - Copy files to all nodes
    - Execute commands on all or a particular node
    - Test rsh connectivity and authorization
    - Test TCP/IP networking connectivity
    - Reboot the entire cluster or specific nodes
    - Shutdown nodes or the cluster in entirety

Installation, Configuration and Testing

- Each computer system to be network booted to execute low-level memory tests for 12+ hours
- Once passed, Operating Systems to be loaded and Linux stress tests to be executed for 24+ hours
    - Tests to include processor and memory intensive applications that have been shown to cause faults in the field.
    - A separate set of tests should access all sectors on each hard drive and run filesystem-intensive applications to ensure drive and filesystem reliability.
- Cluster management software, utilities, compilers, libraries and applications to be installed and configured
    - Cluster applications to be executed for 48+ hours in order to insure the final network, hardware and software configurations are stable and reliable.

Appendix II - Application Source Code

```
/*
*       bestDistance.cpp
*       Application that computes the best bands that minimize the distance
*       between one vector and a group of vectors (i.e. between the ground truth
*       target and the sampled truth vectors)
*
*@authors Stefan A. Robila (original code and algorithms), Gerald R. Busardo (Conversion from Java to C++ with MPICH2 and threading support)
*@version 1.0
*@date 9/1/2010
*
*/
#include <mpi.h>
#include <math.h>
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
using namespace std;

// numvectors is the number of vectors in the spectra file
#define NUMVECTORS 5

// vecsize is the number of elements within each vector provided within the spectra file; set to 147 by default
int VECSIZE=147;

// TPN is Threads Per Node, which can be overridden at runtime
int TPN = 8;

// x is an array that holds each vector from the spectra file; the vector will have "vecsize" number of elements
std::vector< vector<double> > x(NUMVECTORS, vector<double>(VECSIZE));

// NUMJOBS holds the default number of jobs we will run
int NUMJOBS=2;

// INTERVALS is an array that holds the start and end intervals for each job that will be run, along with their best bands and min distance
std::vector< vector<std::string> > INTERVALS(5000000, vector<std::string>(5));

// total number of intervals, jobs per master and jobs per slave
int totint=0, JPM, JPS;

template <typename T>
std::string to_string(T const& value) {
            stringstream sstr;
            sstr << value;
            return sstr.str();
}
// struct used to hold arguments passed to computation method (pthread requires single argument for any function calls)
```

```
struct intervalStruct {
          int currentThread;
          int currentInt;
          std::string st;
          std::string en;
};

// generate the number of jobs used to calculate a particular pair of intervals, based upon user input (2^input-1 = number of "jobs")
int generateJobs(char* MODE) {
          //note that jobs is the power of two for jobs
          std::string tail_of_zeros="";

          int i=VECSIZE-NUMJOBS;
          while (i>0) {
                    tail_of_zeros+='0';
                    i--;
          }

          //create the intervals
          std::string start_all="";
          std::string end_all="";
          i=NUMJOBS;
          while (i>0) {
                    start_all+='0';
                    end_all+='1';
                    i--;
          }

          std::string cStart=start_all;
          int intnum=0;

          while (!cStart.compare(end_all)==0) {
                    //get the next band combination

                    char * cdata;
                    cdata = new char [cStart.size()+1];
                    strcpy (cdata, cStart.c_str());

                    int j = cStart.size()-1;
                    bool done = false;
                    while (j>=0 && !done) {
                              if (cdata[j] == '1'){
                                        cdata [j] = '0';
                                        j--;
                              }
                              else {
                                        cdata[j] = '1';
                                        done = true;
                              }
                    }
                    std::string cEnd = cdata;

                    if ( cEnd.compare(cStart)>=0 && (cEnd.size()+tail_of_zeros.size()) == VECSIZE &&
                              (cStart.size()+tail_of_zeros.size()) == VECSIZE ) {
                              INTERVALS[intnum][0]=cStart+tail_of_zeros;
```

```
                        INTERVALS[intnum][1]=cEnd+tail_of_zeros;
                        if (strcmp(MODE,"OFF")==0) {
                            cout << "Job #" << intnum+1 << ": " << INTERVALS[intnum][0] << " " << INTERVALS[intnum][1] << endl;
                        }
                        intnum++;
                        cStart = cEnd;
                }
                else
                {
                    std::cout << "Error in job creation; End is not greater than Start or they are not equal to VECSIZE" <<endl << endl;
                    std::cout << "Start: " << cStart << endl << "End: " << cEnd << endl << "cStart size: " << cStart.size()
                                << endl << "cEnd size: " << cEnd.size() << endl;
                        return(1);
                }
        }

        totint=intnum;
        cout << endl;
        return(0);
}

void *computeBestDistance(void *structArg){
        int i,j, currentInterval, cthread;
        std::string current;
        std::string end;
        std::string result;
        char * cdata;
        double min_distance=10000;
        std::string best_distance_bands="";
        double current_distance=0;

        struct intervalStruct *currentInput;
        currentInput = (struct intervalStruct *)structArg; // type cast to a pointer to the struct

        cthread = currentInput->currentThread;
        currentInterval = currentInput->currentInt;
        current = currentInput->st;
        end = currentInput->en;

        while (end.compare(current)>=0) {
                int numbands=0;

                // compute the norms (used in the lower part of the distance)
                for (j=0; j<VECSIZE; j++) {
                        if (current.at(j)=='1')
                                numbands++;
                }

                if (numbands >= 2){
                        double distances[NUMVECTORS];
                        double norms[NUMVECTORS];

                        //compute the norms (used in the lower part of the distance)
                        for (i=0; i<NUMVECTORS; i++) {
                                norms[i]=0;
                                for (j=0; j<VECSIZE; j++) {
                                    if (current.at(j)=='1')
                                        norms[i] += ((x[i][j])*(x[i][j]));
                                }
                                norms[i]=sqrt(norms[i]);
                        }

                        //compute the distances
                        current_distance = 0;
                        for (i=1; i<NUMVECTORS; i++){
                                distances[i]=0;
                                for (j=0; j<VECSIZE; j++) {
                                    if (current.at(j)=='1')
                                        distances[i] += x[0][j]*x[i][j];
                                }
                                distances[i]=distances[i]/(norms[0]*norms[i]);
                                current_distance += acos(distances[i]);
                        }
                        current_distance /= (NUMVECTORS-1);

                        // see if this is a better band combination
                        if (current_distance < min_distance){
                                min_distance = current_distance;
                                best_distance_bands = current;
                        }

                }

                //get the next band combination
                cdata = new char [current.size()+1];
                strcpy (cdata, current.c_str());

                j = VECSIZE-1;
                bool done = false;
                while (j>=0 && !done) {
                        if (cdata[j] == '1'){
                                cdata [j] = '0';
                                j--;
                        }
                        else {
                                cdata[j] = '1';
                                done = true;
                        }
                }
                current = cdata;
                free(cdata);
        }
        // save the results
        INTERVALS[currentInterval][2]=best_distance_bands;
        INTERVALS[currentInterval][3]=to_string(min_distance);

        if (TPN>1) {
                pthread_exit(NULL);
        }
}
```

```
void readfileinVectors(char *filename) {
        // declare string buffer
        std::string buffer;
        ifstream infile(filename);


        // get line from file
        getline(infile, buffer, '\n');

        //  read through buffer searching for whitespace or end of string. the start of
        //  each entry is the first element of the string (first word), or the
        //  position after the last whitespace encountered (all other words).  the end
        //  of the entry is the last element of the string (last word), or the
        //  position before the next whitespace (all other words)
        std::string temp;
        unsigned long start=0;
        unsigned long end=0;

        // count is the current element in the vector
        int count=0;

        // v is the vector number from within the spectra file
        int v = 1;
        bool done = false;
        while (end<=buffer.size() && !done) {
                // increment end counter
                end++;

                //  check if position of 'end' in the string is a whitespace, past the end of
                //  the string, or just another character.  if at a non-whitespace character,
                //  skip to the start of the next loop
                if (end<buffer.size() && buffer[end]!=' ') {
                        continue;
                }
                // assign token to a temporary string
                temp.assign(buffer,start,end-start);
                // the line that converts the string to a double and saves it
                x[v][count] = atof(temp.c_str());
                // %vecsize allows the count to go from 1 to 1-vecsize, and then 0
                count=(count+1)%VECSIZE;
                if (count==0){
                        v++;
                        if (v >= NUMVECTORS)
                                done=true;
                }
                // set the start and end to the proper next position
                end++;
                start=end;
        }
        for (count = 0; count<VECSIZE; count++){
                x[0][count]=0;
                for (v=1;v<NUMVECTORS;v++)
                        x[0][count]+=x[v][count];
                x[0][count]/=NUMVECTORS;
        }
```

```
}
int main(int argc, char *argv[]){
        int  numnodes, currentnode, currentInterval, provided, sizeBuf, sizeBuf1, sizeBuf2, complete=1;
        double startwtime = 0.0, endwtime;
        std::string intStartS, intEndS, result;
        char recBuf1[VECSIZE], recBuf2[VECSIZE], sendBuf1[VECSIZE], sendBuf2[VECSIZE], *PMODE="OFF";

        MPI_Init_thread(&argc,&argv,MPI_THREAD_SERIALIZED,&provided);
        MPI_Comm_size(MPI_COMM_WORLD,&numnodes);
        MPI_Comm_rank(MPI_COMM_WORLD,&currentnode);
        MPI_Status status;
        MPI_Request ireq[4*totint];

        //The application is started by five arguments: filename, vecsize, num jobs and performance mode
        if ( argc > 4 ) {
                /***** Initializations *****/
                char *filename = argv[1];
                VECSIZE=atoi(argv[2]);
                NUMJOBS=atoi(argv[3]);
                TPN=atoi(argv[4]);
                if (argc>5) {
                        PMODE=argv[5];
                }

                pthread_t thread[TPN];
                struct intervalStruct intStruct[TPN];

                /***** Master task only ******/
                if (currentnode == 0) {
                        if (strcmp(PMODE,"--performance")==0) {
                                PMODE="ON";
                        }
                        else
                                PMODE="OFF";
                        cout << endl << "Running application with the following parameters:" << endl << endl << "Filename: " << filename
                                <<endl << "Vector Size: " << VECSIZE << endl << "# jobs: " << NUMJOBS << endl << "# of nodes: "
                                << numnodes << endl << "# of threads per node: " << TPN << endl << "Performance mode: " << PMODE
                                << endl << endl;

                        // check inputs
                        if (VECSIZE < NUMJOBS) {
                                std::cout << "Incorrect use, num bands smaller than power num intervals" << endl;
                                MPI_Finalize();
                                return(1);
                        }

                        //Need to read vector data here
                        try {
                                readfileinVectors(filename);
                        } catch (char * e) {
                                std::cout << "Caught IOException: " << e << endl;
                                MPI_Finalize();
                                return(1);
                        }
```

```
            // generate jobs / get st and end
            if (generateJobs(PMODE)==1)
            {
                        std::cout << "Caught Program Exception during job creation... quitting program" << endl;
                        MPI_Finalize();
                        return(1);
            }

            // JPN is auto-calculated

            // determine spread of jobs
            if (totint % numnodes == 0) {
                        JPS = totint/numnodes;
                        JPM = JPS;
                        cout << "Assigning " << JPS << " intervals to both the slave and master nodes." << endl;
            }

            if (totint % numnodes != 0) {
                        JPS = (totint-totint%numnodes)/numnodes;
                        JPM = JPS+(totint%numnodes);
                        cout << "Assigning " << JPS << " intervals per slave and " << JPM << " intervals to the master." << endl;
            }

            cout << endl << "Starting timer and beginning calculations. Please wait... " << endl << endl;

            // send data to slaves - create new local array that holds the intervals
            // broadcast important variables to all nodes
            MPI_Bcast(&JPM,1,MPI_INT,0,MPI_COMM_WORLD);
            MPI_Bcast(&JPS,1,MPI_INT,0,MPI_COMM_WORLD);
            MPI_Bcast(&TPN,1,MPI_INT,0,MPI_COMM_WORLD);
            MPI_Bcast(&VECSIZE,1,MPI_INT,0,MPI_COMM_WORLD);
            MPI_Bcast(&NUMJOBS,1,MPI_INT,0,MPI_COMM_WORLD);
            MPI_Bcast(&PMODE,1,MPI_CHAR,0,MPI_COMM_WORLD);

            // send intervals
            for (int dest=1; dest<numnodes; dest++) {
                        currentInterval=JPM+(JPS*dest)-JPS;

                        for (int i1=0; i1<JPS; i1++) {
                                    sprintf(sendBuf1, INTERVALS[currentInterval][0].c_str());
                                    sprintf(sendBuf2, INTERVALS[currentInterval][1].c_str());
                                    sizeBuf=strlen(sendBuf1)+1;

                                    MPI_Send(&sizeBuf,1, MPI_INT, dest, 3, MPI_COMM_WORLD);
                                    MPI_Send(sendBuf1,sizeBuf, MPI_CHAR, dest, 1+dest, MPI_COMM_WORLD);
                                    MPI_Send(sendBuf2,sizeBuf, MPI_CHAR, dest, 2+dest, MPI_COMM_WORLD);

                                    currentInterval++;
                        }
            }
            // master to start timer
            startwtime = MPI_Wtime();
}
```

```
// **************************************************
MPI_Barrier( MPI_COMM_WORLD ) ;
// **************************************************


// ***** Non-master tasks only *****
if (currentnode>0) {
        //Need to read vector data here
        try {
                readfileinVectors(filename);
        } catch (char * e) {
                std::cout << "Caught IOException: " << e << endl;
                MPI_Finalize();
                return(1);
        }

        // receive broadcast of important variables
        MPI_Bcast(&JPM,1,MPI_INT,0,MPI_COMM_WORLD);
        MPI_Bcast(&JPS,1,MPI_INT,0,MPI_COMM_WORLD);
        MPI_Bcast(&TPN,1,MPI_INT,0,MPI_COMM_WORLD);
        MPI_Bcast(&VECSIZE,1,MPI_INT,0,MPI_COMM_WORLD);
        MPI_Bcast(&NUMJOBS,1,MPI_INT,0,MPI_COMM_WORLD);
        MPI_Bcast(&PMODE,1,MPI_CHAR,0,MPI_COMM_WORLD);

        currentInterval=JPM+(JPS*currentnode)-JPS;
        int currentThread=0;
        int i4=0;

        while (i4 < JPS) {
                while (currentThread<TPN && i4<JPS) {
                        MPI_Recv(&sizeBuf,1, MPI_INT,0,3,MPI_COMM_WORLD, &status);
                        MPI_Recv(recBuf1,sizeBuf, MPI_CHAR,0,1+currentnode,MPI_COMM_WORLD, &status);
                        MPI_Recv(recBuf2,sizeBuf, MPI_CHAR,0,2+currentnode,MPI_COMM_WORLD, &status);

                        intStruct[currentThread].currentThread = currentThread;
                        intStruct[currentThread].currentInt = currentInterval;
                        intStruct[currentThread].st = recBuf1;
                        intStruct[currentThread].en = recBuf2;

                        if (TPN>1) {
                            /* Create independent threads each of which will execute function */
                          if (pthread_create(&thread[currentThread], NULL, computeBestDistance, (void *)
                                                                                    &intStruct[currentThread])) {
                                  cout << stderr << "Error while creating thread..." << endl;
                                  MPI_Finalize();
                                  return(1);
                          } else
                                  currentThread++;
                        } else
                          computeBestDistance(&intStruct);
                          currentInterval++;
                          i4++;
                }
                if (TPN>1) {
                        for (int j=currentThread-1;j>0;j--) {
```

```
                                        pthread_join(thread[j], NULL);
                              }
                    }
                    currentThread=0;
          }
}

if (currentnode == 0) {
          // master to perform it's work
          currentInterval = 0;
          int currentThread=0;

          while (currentInterval < JPM) {
                    while (currentThread<TPN && currentInterval<JPM){
                              intStruct[currentThread].currentThread = currentThread;
                              intStruct[currentThread].currentInt = currentInterval;
                              intStruct[currentThread].st = INTERVALS[currentInterval][0];
                              intStruct[currentThread].en = INTERVALS[currentInterval][1];

                              if (TPN>1) {
                                        /* Create independent threads each of which will execute function */
                                        if (pthread_create(&thread[currentThread], NULL, computeBestDistance, (void *)
                                                                        &intStruct[currentThread])) {
                                                  cout << stderr << "Error while creating thread..." << endl;
                                                  MPI_Finalize();
                                                  return(1);
                                        } else
                                                  currentThread++;
                              } else
                                        computeBestDistance(&intStruct);
                              currentInterval++;
                    }

                    if (TPN>1) {
                              for (int j=currentThread-1;j>0;j--) {
                                        pthread_join(thread[j], NULL);
                              }

                              currentThread=0;
                    }
          }
}

MPI_Barrier( MPI_COMM_WORLD ) ;
if (currentnode == 0) {
          // master to calculate and print out end time
          endwtime = MPI_Wtime();
}

if (currentnode > 0) {
          if (strcmp(PMODE,"OFF")==0) {
                    currentInterval=JPM+(JPS*currentnode)-JPS;

                    for (int i4 = 0; i4 < JPS; i4++) {
                              // send the results back to the master node
```

```
                                                sprintf(sendBuf1, INTERVALS[currentInterval][2].c_str());
                                                sizeBuf1=strlen(sendBuf1)+1;
                                                sprintf(sendBuf2, INTERVALS[currentInterval][3].c_str());
                                                sizeBuf2=strlen(sendBuf2)+1;

                                                MPI_Send(&sizeBuf1,1, MPI_INT, 0, 8+currentnode, MPI_COMM_WORLD);
                                                MPI_Send(&sizeBuf2,1, MPI_INT, 0, 9+currentnode, MPI_COMM_WORLD);
                                                MPI_Send(sendBuf1,sizeBuf1, MPI_CHAR, 0, 5+currentnode, MPI_COMM_WORLD);
                                                MPI_Send(sendBuf2,sizeBuf2, MPI_CHAR, 0, 6+currentnode, MPI_COMM_WORLD);

                                                currentInterval++;
                                        }
                                        MPI_Irecv(&complete,1, MPI_INT, 0, 10, MPI_COMM_WORLD, ireq);
                                }
                        }

                        if (currentnode == 0) {
                                if (strcmp(PMODE,"OFF")==0) {
                                        // master to collect results
                                        cout << "Timer stopped and collecting results... " << endl << endl;

                                        for (int dest=1; dest<numnodes; dest++) {
                                                currentInterval=JPM+(JPS*dest)-JPS;

                                                for (int i5=0; i5<JPS; i5++) {
                                                   MPI_Recv(&sizeBuf1,1, MPI_INT,dest,8+dest,MPI_COMM_WORLD, &status);
                                                   MPI_Recv(&sizeBuf2,1, MPI_INT,dest,9+dest,MPI_COMM_WORLD, &status);
                                                   MPI_Recv(recBuf1,sizeBuf1, MPI_CHAR, dest, 5+dest, MPI_COMM_WORLD, &status);
                                                   MPI_Recv(recBuf2,sizeBuf2, MPI_CHAR, dest, 6+dest, MPI_COMM_WORLD, &status);

                                                   INTERVALS[currentInterval][2]=recBuf1;
                                                   INTERVALS[currentInterval][3]=recBuf2;
                                                   currentInterval++;
                                                }
                                                MPI_Isend(&complete,1, MPI_INT, dest, 10, MPI_COMM_WORLD, ireq);
                                                MPI_Wait(ireq,&status);
                                        }

                                        for (int i5 = 0; i5 < totint; i5++) {
                                                cout << "Job #" << i5+1 << ": " << INTERVALS[i5][0] << " " << INTERVALS[i5][1] <<
                                                        " :: Best Distance Bands: " << INTERVALS[i5][2] << " Min Distance: " <<
                                                        INTERVALS[i5][3] << endl;
                                        }
                                }
                                cout << endl << "Performance --> start time: " << startwtime << " end time: " << endwtime << " :: total time (s): " <<
                                        endwtime-startwtime << " total time (m): " << (endwtime-startwtime)/60 << endl << endl;
                        }
                }
                else
                {
                        std::cout << "ERROR - the application is started by four arguements: filename, vecsize, num jobs and threads per node"<< endl;
                        MPI_Finalize();
                        return(1);
                }
                MPI_Finalize();
```

```
                    return(0);
}
```

An example of the output without the –*performance* command (abbreviated due to length):

```
[busardog@copou final]$ mpirun  -n 65 ./bestDistance spectra_1.txt 13 12 16

Running application with the following parameters:

Filename: spectra_1.txt
Vector Size: 13
# jobs: 12
# of nodes: 65
# of threads per node: 16
Performance mode: OFF

Job #1: 0000000000000 0000000000010
Job #2: 0000000000010 0000000000100
Job #3: 0000000000100 0000000000110
Job #4: 0000000000110 0000000001000
Job #5: 0000000001000 0000000001010
Job #6: 0000000001010 0000000001100
Job #7: 0000000001100 0000000001110
Job #8: 0000000001110 0000000010000
Job #9: 0000000010000 0000000010010
Job #10: 0000000010010 0000000010100
Job #11: 0000000010100 0000000010110
Job #12: 0000000010110 0000000011000
Job #13: 0000000011000 0000000011010
Job #14: 0000000011010 0000000011100
Job #15: 0000000011100 0000000011110
<snip>
Job #4088: 1111111101110 1111111110000
Job #4089: 1111111110000 1111111110010
Job #4090: 1111111110010 1111111110100
Job #4091: 1111111110100 1111111110110
Job #4092: 1111111110110 1111111111000
Job #4093: 1111111111000 1111111111010
Job #4094: 1111111111010 1111111111100
Job #4095: 1111111111100 1111111111110

Assigning 63 intervals to both the slave and master nodes.

Starting timer and beginning calculations. Please wait...

Timer stopped and collecting results...

Job #1: 0000000000000 0000000000010 :: Best Distance Bands:  Min Distance: 10000
Job #2: 0000000000010 0000000000100 :: Best Distance Bands: 0000000000011 Min Distance: 0.050296
Job #3: 0000000000100 0000000000110 :: Best Distance Bands: 0000000000110 Min Distance: 0.0156177
Job #4: 0000000000110 0000000001000 :: Best Distance Bands: 0000000000110 Min Distance: 0.0156177
Job #5: 0000000001000 0000000001010 :: Best Distance Bands: 0000000001010 Min Distance: 0.0290525
Job #6: 0000000001010 0000000001100 :: Best Distance Bands: 0000000001010 Min Distance: 0.0290525
Job #7: 0000000001100 0000000001110 :: Best Distance Bands: 0000000001110 Min Distance: 0.0328368
Job #8: 0000000001110 0000000010000 :: Best Distance Bands: 0000000001110 Min Distance: 0.0328368
Job #9: 0000000010000 0000000010010 :: Best Distance Bands: 0000000010010 Min Distance: 0.0338363
Job #10: 0000000010010 0000000010100 :: Best Distance Bands: 0000000010010 Min Distance: 0.0338363
Job #11: 0000000010100 0000000010110 :: Best Distance Bands: 0000000010110 Min Distance: 0.0370364
Job #12: 0000000010110 0000000011000 :: Best Distance Bands: 0000000011000 Min Distance: 0.0109012
Job #13: 0000000011000 0000000011010 :: Best Distance Bands: 0000000011000 Min Distance: 0.0109012
Job #14: 0000000011010 0000000011100 :: Best Distance Bands: 0000000011010 Min Distance: 0.0318162
Job #15: 0000000011100 0000000011110 :: Best Distance Bands: 0000000011110 Min Distance: 0.0377892
<snip>
Job #4088: 1111111101110 1111111110000 :: Best Distance Bands: 1111111101111 Min Distance: 0.115334
Job #4089: 1111111110000 1111111110010 :: Best Distance Bands: 1111111110001 Min Distance: 0.112641
Job #4090: 1111111110010 1111111110100 :: Best Distance Bands: 1111111110011 Min Distance: 0.113907
Job #4091: 1111111110100 1111111110110 :: Best Distance Bands: 1111111110101 Min Distance: 0.115644
Job #4092: 1111111110110 1111111111000 :: Best Distance Bands: 1111111111000 Min Distance: 0.11363
Job #4093: 1111111111000 1111111111010 :: Best Distance Bands: 1111111111001 Min Distance: 0.109869
Job #4094: 1111111111010 1111111111100 :: Best Distance Bands: 1111111111011 Min Distance: 0.110637
Job #4095: 1111111111100 1111111111110 :: Best Distance Bands: 1111111111101 Min Distance: 0.112143

Performance --> start time: 1.28807e+09 end time: 1.28807e+09 :: total time (s): 0.0123601 total time (m): 0.000206002
```

An example of the output with the –*performance* command:

```
 [busardog@copou final]$ mpirun  -n 65 ./bestDistance spectra_1.txt 13 12 16 --performance

Running application with the following parameters:

Filename: spectra_1.txt
Vector Size: 13
# jobs: 12
# of nodes: 65
# of threads per node: 16
Performance mode: ON

Assigning 63 intervals to both the slave and master nodes.

Starting timer and beginning calculations. Please wait...

Performance --> start time: 1.28807e+09 end time: 1.28807e+09 :: total time (s): 0.017467 total time (m): 0.000291117
```

Appendix III – Leveraging MPI for Parallelization of C/C++ Code

There is much documentation regarding MPI on the Internet. What follows is only the basic components of MPI utilized as part of this project. More details can be found in [**17**] and [**18**].

Installation

Installation of MPI depends on the version of MPI used, as well as the system MPI is being installed on. For details on how to install MPICH on a typical, Linux-based Beowulf cluster, see Appendix V – Technical Setup Details for a Beowulf Cluster.

Usage

MPI usage can be as simple or complex as one would like it to be. Yet, one can typically accomplish all that they need via just a few basic commands. To get started, one must first include the necessary MPI libraries via:

```
#include <mpi.h>
```

Then, MPI must be initialized:

```
MPI_Init_thread(&argc,&argv,MPI_THREAD_SERIALIZED,&provided);
```

where *argc* and *argv* are the arguments passed to main, M*PI_THREAD_SERIALIZED* and *provided* define the type of threading that is allowed.

Next, several inquiry routines can be called:

```
MPI_Comm_size(MPI_COMM_WORLD,&numnodes);
MPI_Comm_rank(MPI_COMM_WORLD,&currentnode);
```

where *MPI_Comm_size* saves within a variable (numnodes) the number of nodes in use at runtime and *MPI_Comm_*rank saves within a variable (currentnode) the current node that is executing the code.

To restrict the execution of code to a particular node, you can use the variables defined by the above-mentioned inquiry routines, as such:

```
if (currentnode == 0) {
        // perform code that only should be executed on the master node

}


if (currentnode > 0) {

        // perform code that only should be executed on slave nodes

}
```

To broadcast data to all nodes, the following command must be run on all nodes, not just the node that is to send the data:

```
MPI_Bcast(&JPM,1,MPI_INT,0,MPI_COMM_WORLD);
```

where *JPM* is the variable you wish to send, 1 is the number of variables you are passing (if passing an array, this should be the total size of the array), MPI_INT is the MPI datatype of the variable being sent, 0 is the node that currently holds the variable which you wish to broadcast and MPI_COMM_WORLD is the communications group previously definied during initialization.

To send data to a specific node, a pair of MPI_Send and MPI_Receive commands must be run, such that the MPI_Send is exsecuted on the node that is to send the data and MPI_Receive is executed on the node that is to receive the data. The syntax for a Send is:

```
MPI_Send(&sizeBuf,1, MPI_INT, dest, 3, MPI_COMM_WORLD);
```

where *sizeBuf* is the variable to be sent, 1 is the count of variables to be sent, MPI_INT is the MPI datatype of the variable being sent, 3 is the destination node and MPI_COMM_WORLD is the communications group. The syntax for an associated Receive is:

```
MPI_Recv(&sizeBuf,1, MPI_INT,0,3,MPI_COMM_WORLD, &status);
```

where *sizeBuf* is the variable to be received, 1 is the total number of variables being sent, MPI_INT is the datatype of the variable being sent, 0 is the node sending the data, 3 is the node receiving the data, MPI_COMM_WORLD is the communications group and *status* holds the result of the send/receive.

To have a node wait until all other nodes reach that spot in the code:

```
MPI_Barrier( MPI_COMM_WORLD ) ;
```

where MPI_COMM_WORLD defines the communications group that holds all nodes that must reach that point in the code before any node can proceed.

Once all work is done, MPI must be finalized via the following command:

```
MPI_Finalize();
```

Compilation

To compile code that leverages MPI, one must simply use the appropriate MPI wrapper compiler (e.g. mpif77 for Fortram, mpicc for C and mpic++ for C++) in place of the regular compiler (e.g. f77, gcc, g++). An example:

```
mpic++  ./bestDistance.cpp –o bestDistance
```

Execution

In order to execute an application that leverages MPI, one must use *mpirun.* Explain usage:

```
mpirun –n 64 ./bestDistance spectra_1.txt 20 12 16
```

where “-n 64” instructs MPI to spread the work across 64 nodes, and “*./bestDistance spectra_1.txt 20 12 16”* is the actual executable and its run-time parameters.

Example of how to run the MPI program “in the background”:

```
mpirun –n 64 ./bestDistance spectra_1.txt 20 12 16 > output-512-2 < /dev/null &
```

where "> *output-512-2*" sends the console output to a textfile, "< */dev/null*" instructs mpirun that there will be no input (mandatory if running an MPI program "in the background"), and the trailing *&* sending the job into the background.

Should one have code that does not leverage a threading library for symmetric multiprocessing on a single node, the following would spread the work across all defined nodes as well as additional cores per node:

```
mpirun -machinefile computeNodes -1 -np 512 ./xhpl > output-512-2 < /dev/null &
```

where "*-machinefile computeNodes*" defines the nodes to be used (typically a subset of the default node list which contains all nodes, the "*-1*" instructs MPI to not perform any work on the current, master node, and "*-np 512"* is mandatory and instructs MPI to spawn 512 individual threads, spread out amongst all nodes in computeNodes. An important note: computeNodes will have to have multiple entries of each node up to a max of the number of potential simultaneous executions. An example if a two-node cluster was to be used where each node has 4 cores:

computeNodes file contents:

```
node0
node0
node0
node0
node1
node1
node1
node1
```

Sample mpirun syntax:

```
mpirun –machinefile computeNodes -1 –np 8 ./programFile
```

where eight individual threads will be spread over two nodes, but 4 per node. For an alternative approach if each node has multiple cores, see Appendix IV – Leveraging "pthreads" for Symmetric Multiprocessing (SMP).

Appendix IV – Leveraging "pthreads" for Symmetric Multiprocessing (SMP)

There are typically two ways to use MPI with multicore processors or multiprocessor nodes: [**36**]

1) Use one MPI process per core (here, a core is defined as a program counter and some set of arithmetic, logic, and load/store units). To enable multiple
2) Use one MPI process per node (here, a node is defined as a collection of cores that share a single address space). Use threads or compiler-provided parallelism to exploit the multiple cores.

With that said, we decided to take the second of the two approaches. To obtain threading capability outside of MPI, we implemented pthreads. While there is much documentation on the web for pthreads, what follows are some details regarding the use of pthreads for the purposes of this project.

First, we must include the pthread library:

```
#include <pthread.h>
```

Then, we must determine which function call we wish to spawn into new threads. In our case, we chose the computeBestDistance function. Since comptueBestDistance took more than one argument, a struct needed to be created in order to abide by the syntax of pthreads.

```
struct intervalStruct {
        int currentThread;
        int currentInt;
        std::string st;
        std::string en; };
```

We then needed to create a few variables to manage the threads:

```
pthread_t thread[TPN];
int currentThread;
```

where thread is an array with a size that matches the total number of simultaneous threads per node.

We then called computeBestDistance in the following way:

```
        pthread_create(&thread[currentThread], NULL, computeBestDistance, (void *)
&intStruct[currentThread])
```

where thread number *currentThread* is stored in *thread* and *computeBestDistance* is called and passed *intStruct*, the struct created to hold serveral variables that *computeBestDistance* requires. Once the number of threads reaches the predetermined maximum as defined via a command-line argument, no additional threads are spawned until all threads' work has completed. This approach was decided to be acceptable since all threads typically will complete at the same time. Once the threads are completed, we collect the results via a loop from

```
pthread_join(thread[j], NULL);
```

where *j* decrements from the last thread number generated to the first. Once the results are collected, a new batch of threads are spawned, if/while work is remaining for that node.

Appendix V – Technical Setup Details for a Beowulf Cluster

The steps required to create a mock 4-node Beowulf cluster using Virtualization on a single PC or Laptop are as follows (adapted from https://help.ubuntu.com/community/MpichCluster):

1) Obtain The Necessary Software
    a. A Hypervisor:
        i. VMware Workstation 7 running on Windows 7
    b. Guest OS (Compute and Slave Node) Media
        i. ISO file for Ubuntu 10.0.4
    c. Message Passing Interface
        i. Mpich2
2) Install and Configure VMware
    a. Setup networking within Virtual Network Editor
        i. Choose a network and set it to NAT
        ii. Define the private subnet
        iii. Edit NAT settings and set Gateway IP (a private IP on the new subnet that represents the VMware "host")
        iv. Determine the IPs that you will use for the nodes; enable DHCP and set a range that does not conflict with the nodes' IPs
    b. Using the Ubuntu ISO file, create a Linux VM using the default VMware configuration parameters;
        i. we will call the host ub0; use DHCP for network IP for now
        ii. IMPORTANT: Enable vmci so that VMs can communicate with each other
3) Begin Configuring Master (ub0)
    a. Power on the master node and set static IP to the one chosen earlier; gateway should be the one set within the Virtual Network Editor under NAT settings
    b. Begin defining hostnames in etc/hosts

```
127.0.0.1      localhost
192.168.133.100 ub0
192.168.133.101 ub1
192.168.133.102 ub2
192.168.133.103 ub3
```

4) Install NFS
    a. NFS allows us to create a folder on the master node and have it synced on all the other nodes. This folder can be used to store the code that ultimately will be run using MPI. To Install NFS just run this in the master node's terminal:

```
omid@ub0:~$ sudo apt-get install nfs-kernel-server
```

5) Sharing Master Folder
    a. Make a folder in all nodes; this is where we'll store our data and programs

```
omid@ub0:~$ sudo mkdir /mirror
```

b. Then, to share the contents of this folder located on the master node to all the other nodes, edit the /etc/exports file on the master node to contain the additional line

/mirror *(rw,sync)

6) Shut down the master VM and clone it; power up the clone and rename it to ub1 and set the static IP; do the same for ub2 and ub3

7) Power up master node

8) On each slave node (ub1-ub3), mount /master
    a. To mount the folder on the other nodes, change fstab in order to mount it on every boot. We do this by editing /etc/fstab and adding this line:

ub0:/mirror /mirror nfs

9) On each node, including master, define a user with same name and same userid with a home directory in /mirror.

10) Change the owner of /mirror to mpiu:

```
omid@ub0:~$ sudo chown mpiu /mirror
```

11) Install SSH Server
    a. Run this command in all nodes in order to install OpenSSH Server

```
omid@ub0:~$ sudo apt-get install openssh-server
```

    b. Set up SSH with no pass phrase for communication between nodes by first logging in with our new user:

```
omid@ub0:~$ su – mpiu
```

    c. Then generate DSA key for mpiu:

```
mpiu@ub0:~$ ssh-keygen -t dsa
```

Leave passphrase empty.

    d. Next we add this key to authorized keys:

```
mpiu@ub0:~$ cd .ssh
mpiu@ub0:~/.ssh$ cat id_pub.dsa >> authorized_keys2
```

    e. Note: As the home directory of mpiu in all nodes is the same (/mirror/mpiu) , there is no need to run these commands on all nodes.

    f. To test SSH run:

```
mpiu@ub0:~$ ssh ub1 hostname
```

    g. It should return remote hostname without asking for passphrase.

12) Installing GCC

a. Install build-essential package:

```
mpiu@ub0:~$ sudo apt-get install build-essential
```

13) Installing MPICH2

a. Download MPICH2 source code from http://www-unix.mcs.anl.gov/mpi/mpich .

b. Extract .tar.bz2 file in /mirror. Also make a folder for MPICH installation.

```
mpiu@ub3:/mirror$ mkidr mpich2
mpiu@ub3:/mirror$ tar xvf mpich2-1.0.5p3.tar.gz
mpiu@ub3:/mirror$ cd mpich2-1.0.5p3
mpiu@ub3:/mirror/mpich2-1.0.5p3$ ./configure --prefix=/mirror/mpich2
mpiu@ub3:/mirror/mpich2-1.0.5p3$ make
mpiu@ub3:/mirror/mpich2-1.0.5p3$ sudo make install
```

c. After successfully compiling and installing mpich, add these lines to "/mirror/mpiu/.bashrc/"

```
export PATH=/mirror/mpich2/bin:$PATH
export PATH
LD_LIBRARY_PATH="/mirror/mpich2/lib:$LD_LIBRARY_PATH"
export LD_LIBRARY_PATH
```

d. Next we run this command in order to define MPICH installation path to SSH.

```
mpiu@ub0:~$ sudo echo /mirror/mpich2/bin >> /etc/environment
```

e. For testing our installation run:

```
mpiu@ub0:~$  which mpd
mpiu@ub0:~$  which mpiexec
mpiu@ub0:~$  which mpirun
```

14) Setting up MPD

a. Create mpd.hosts in mpiu's home directory with nodes names:

```
ub3
ub2
ub1
ub0
```

b. Run :

```
mpiu@ub0:~$ echo secretword=something   >> ~/.mpd.conf
mpiu@ub0:~$ chmod 600 .mpd.conf
```

c. To test MPD run the following ommands. The output should be the current hostname.

```
mpiu@ub0:~$ mpd &
mpiu@ub0:~$ mpdtrace
mpiu@ub0:~$ mpdallexit
```

d. Run mpd daemon:

```
mpiu@ub0:~$ mpdboot -n 4
mpiu@ub0:~$ mpdtrace
```

e. Confirm that the output now is the name of all nodes.
    i. If this doesn't succeed try running mpdcheck on all hosts to find possible errors in conf files (they will be marked with **).

Appendix VI –Tuning of the HPL Benchmark

Installation

The HPL benchmark requires several prerequisite libraries before compilation of the benchmark itself, with the most important of all being a tuned Basic Linear Algebra Subprograms (BLAS) library. [**13**]

To obtain a tuned BLAS, we downloaded and ran Automatically Tuned Linear Algebra Subprograms (ATLAS). [**37**] Following the unpacking of the compressed tarball, the makefile was created via the following command:

make config CC=mpicc

The config then prompts you for input and provides instructions as needed in order to obtain all of the necessary environmental characteristics including default compilers and flags. What follows is the resulting configuration file:

```
#  ----------------------------
#  Make.ARCH for ATLAS3.6.0
#  ----------------------------

#  ---------------------------------
#  Make sure we get the correct shell
#  ---------------------------------
   SHELL = /bin/sh

#  -----------------------------------------------
#  Name indicating the platform to configure BLAS to
#  -----------------------------------------------
   ARCH = Linux_HAMMER64SSE2_8

#  ------------------
#  Various directories
#  ------------------
   TOPdir = /home/busardog/project/benchmarks/ATLAS
   INCdir = $(TOPdir)/include/$(ARCH)
   SYSdir = $(TOPdir)/tune/sysinfo/$(ARCH)
   GMMdir = $(TOPdir)/src/blas/gemm/$(ARCH)
   UMMdir = $(GMMdir)
   GMVdir = $(TOPdir)/src/blas/gemv/$(ARCH)
   GR1dir = $(TOPdir)/src/blas/ger/$(ARCH)
   L1Bdir = $(TOPdir)/src/blas/level1/$(ARCH)
   L2Bdir = $(TOPdir)/src/blas/level2/$(ARCH)
   L3Bdir = $(TOPdir)/src/blas/level3/$(ARCH)
   TSTdir = $(TOPdir)/src/testing/$(ARCH)
   AUXdir = $(TOPdir)/src/auxil/$(ARCH)
   CBLdir = $(TOPdir)/interfaces/blas/C/src/$(ARCH)
   FBLdir = $(TOPdir)/interfaces/blas/F77/src/$(ARCH)
   BINdir = $(TOPdir)/bin/$(ARCH)
   LIBdir = $(TOPdir)/lib/$(ARCH)
   PTSdir = $(TOPdir)/src/pthreads
   MMTdir = $(TOPdir)/tune/blas/gemm/$(ARCH)
   MVTdir = $(TOPdir)/tune/blas/gemv/$(ARCH)
   R1Tdir = $(TOPdir)/tune/blas/ger/$(ARCH)
   L1Tdir = $(TOPdir)/tune/blas/level1/$(ARCH)
   L3Tdir = $(TOPdir)/tune/blas/level3/$(ARCH)

#  ---------------------------------------------------------------------
#  Name and location of scripts for running executables during tuning
#  ---------------------------------------------------------------------
   ATLRUN = $(BINdir)/ATLrun.sh
   ATLFWAIT = $(BINdir)/xatlas_waitfile

#  ---------------------
#  Libraries to be built
#  ---------------------
   ATLASlib = $(LIBdir)/libatlas.a
   CBLASlib = $(LIBdir)/libcblas.a
   F77BLASlib = $(LIBdir)/libf77blas.a
   PTCBLASlib = $(LIBdir)/libptcblas.a
   PTF77BLASlib = $(LIBdir)/libptf77blas.a
   LAPACKlib = $(LIBdir)/liblapack.a

   TESTlib = $(LIBdir)/libtstatlas.a
#  -------------------------------------------
#  Upper bound on largest cache size, in bytes
#  -------------------------------------------
   L2SIZE = -DL2SIZE=1048576

#  ---------------------------------------
#  Command setting up correct include path
#  ---------------------------------------
   INCLUDES = -I$(TOPdir)/include -I$(TOPdir)/include/$(ARCH) \
              -I$(TOPdir)/include/contrib

#  -------------------------------------------
#  Defines for setting up F77/C interoperation
#  -------------------------------------------
   F2CDEFS = -DAdd_ -DStringSunStyle

#  --------------------------------------
#  Special defines for user-supplied GEMM
#  --------------------------------------
   UMMDEFS =

#  ------------------------------
#  Architecture identifying flags
#  ------------------------------
   ARCHDEFS = -DATL_OS_Linux -DATL_ARCH_HAMMER64 -DATL_SSE2 -DATL_SSE1 -DATL_GAS_x8664 -m64

#  -------------------------------------------------------------------
#  NM is the flag required to name a compiled object/executable
#  OJ is the flag required to compile to object rather than executable
#  These flags are used by all compilers.
#  -------------------------------------------------------------------
   NM = -o
   OJ = -c
```

```
#  ---------------------------------------------------------------------------
#  Fortran 77 compiler and the flags to use.  Presently, ATLAS does not itself
#  use any Fortran 77, but vendor BLAS are typically written for Fortran, so
#  any links that include non-ATLAS BLAS will use FLINKER instead of CLINKER
#  ---------------------------------------------------------------------------
   F77 = mpif77
   F77FLAGS = -fomit-frame-pointer -O -m64
   FLINKER = $(F77)
   FLINKFLAGS = $(F77FLAGS)
   FCLINKFLAGS = $(FLINKFLAGS)


#  ---------------------------------------------------------------------------
#  Various C compilers, and the linker to be used when we are not linking in
#  non-ATLAS BLAS (which usually necessitate using the Fortran linker).
#  The C compilers recognized by ATLAS are:
#     CC :  Compiler to use to compile regular, non-generated code
#    MCC :  Compiler to use to compile generated, highly-optimized code
#    XCC :  Compiler to be used on the compile engine of a cross-compiler
#  These will typically all be the same.  An example of where this is not
#  the case would be DEC ALPHA 21164, where you want to use gcc for MCC,
#  because DEC's cc does not allow the programmer access to all 32 floating
#  point registers.  However, on normal C code, DEC's cc produces much faster
#  code than gcc, so you CC set to cc.  Of course, any system where you are
#  cross-compiling, you will need to set XCC differently than CC & MCC.
#  ---------------------------------------------------------------------------
   CDEFS = $(L2SIZE) $(INCLUDES) $(F2CDEFS) $(ARCHDEFS) -DATL_NCPU=8

   GOODGCC = gcc
   CC = /usr/bin/gcc
   CCFLAG0 = -fomit-frame-pointer -O -mfpmath=387 -m64
   CCFLAGS = $(CDEFS) $(CCFLAG0)
   MCC = /usr/bin/gcc
   MMFLAGS = -fomit-frame-pointer -O -mfpmath=387 -m64
   XCC = /usr/bin/gcc
   XCCFLAGS = $(CDEFS) -fomit-frame-pointer -O -mfpmath=387 -m64
   CLINKER = $(CC)
   CLINKFLAGS = $(CCFLAGS)
   BC = $(CC)
   BCFLAGS = $(CCFLAGS)
   ARCHIVER = ar
   ARFLAGS  = r
   RANLIB   = echo

#  -------------------------------------
#  tar, gzip, gunzip, and parallel make
#  -------------------------------------
   TAR    = /bin/tar
   GZIP   = /bin/gzip
   GUNZIP = /bin/gunzip
   PMAKE  = $(MAKE) -j 8

#  ------------------------------------
#  Reference and system libraries
#  ------------------------------------
```

```
   BLASlib =
   FBLASlib =
   FLAPACKlib =
   LIBS = -lpthread -lm

#  ----------------------------------------------------------
#  ATLAS install resources (include arch default directories)
#  ----------------------------------------------------------
   ARCHDEF =
   MMDEF =
   INSTFLAGS =

#  ---------------------------------------
#  Generic targets needed by all makefiles
#  ---------------------------------------
waitfile:
```

Once config finishes, the compilation and install process is started via the following command:

make install arch=Linux_HAMMER64SSE2_8

where *Linux_HAMMER64SSE2_8* was selected as our architecture during the install process. The total time required for compilation was about an hour.

Once we had a tuned BLAS via ATLAS, we then were able to compile the HPL benchmark. We downloaded HPL 2.0 from its homepage [**13**] and unzipped the tarball in a directly called ‘hpl’. We then created the following Make.Hammer file in the top-level directory:

```
#  -- High Performance Computing Linpack Benchmark (HPL)
#     HPL - 2.0 - September 10, 2008
#     Antoine P. Petitet
#     University of Tennessee, Knoxville
#     Innovative Computing Laboratory
#     (C) Copyright 2000-2008 All Rights Reserved
#
#  -- Copyright notice and Licensing terms:
#
#  Redistribution  and  use in  source and binary forms, with or without
#  modification, are  permitted provided  that the following  conditions
#  are met:
#
#  1. Redistributions  of  source  code  must retain the above copyright
#  notice, this list of conditions and the following disclaimer.
#
#  2. Redistributions in binary form must reproduce  the above copyright
#  notice, this list of conditions,  and the following disclaimer in the
#  documentation and/or other materials provided with the distribution.
#
#  3. All  advertising  materials  mentioning  features  or  use of this
#  software must display the following acknowledgement:
#  This  product  includes  software  developed  at  the  University  of
#  Tennessee, Knoxville, Innovative Computing Laboratory.
#  4. The name of the  University,  the name of the  Laboratory,  or the
#  names  of  its  contributors  may  not  be used to endorse or promote
#  products  derived   from   this  software  without  specific  written
#  permission.
#
#  -- Disclaimer:
#
#  THIS  SOFTWARE  IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
#  ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,  INCLUDING,  BUT NOT
#  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
#  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE UNIVERSITY
#  OR  CONTRIBUTORS  BE  LIABLE FOR ANY  DIRECT,  INDIRECT,  INCIDENTAL,
#  SPECIAL,  EXEMPLARY,  OR  CONSEQUENTIAL DAMAGES  (INCLUDING,  BUT NOT
#  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
#  DATA OR PROFITS; OR BUSINESS INTERRUPTION)  HOWEVER CAUSED AND ON ANY
#  THEORY OF LIABILITY, WHETHER IN CONTRACT,  STRICT LIABILITY,  OR TORT
#  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
#  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
# ######################################################################
#
# ----------------------------------------------------------------------
# - shell --------------------------------------------------------------
# ----------------------------------------------------------------------
#
SHELL        = /bin/sh
#
CD           = cd
CP           = cp
LN_S         = ln -s
MKDIR        = mkdir
RM           = /bin/rm -f
TOUCH        = touch
#
# ----------------------------------------------------------------------
# - Platform identifier ------------------------------------------------
# ----------------------------------------------------------------------
#
ARCH         = Hammer
#
# ----------------------------------------------------------------------
# - HPL Directory Structure / HPL library ------------------------------
# ----------------------------------------------------------------------
#
TOPdir       = $(HOME)/project/benchmarks/hpl
INCdir       = $(TOPdir)/include
BINdir       = $(TOPdir)/bin/$(ARCH)
LIBdir       = $(TOPdir)/lib/$(ARCH)
#
HPLlib       = $(LIBdir)/libhpl.a
#
# ----------------------------------------------------------------------
# - Message Passing library (MPI) --------------------------------------
# ----------------------------------------------------------------------
# MPinc tells the  C  compiler where to find the Message Passing library
# header files,  MPlib  is defined  to be the name of  the library to be
# used. The variable MPdir is only used for defining MPinc and MPlib.
#
MPdir        = /usr/local/mpich2
MPinc        = -I$(MPdir)/include
MPlib        = $(MPdir)/lib/libmpich.a
#
# ----------------------------------------------------------------------
# - Linear Algebra library (BLAS or VSIPL) -----------------------------
# ----------------------------------------------------------------------
# LAinc tells the  C  compiler where to find the Linear Algebra  library
# header files,  LAlib  is defined  to be the name of  the library to be
# used. The variable LAdir is only used for defining LAinc and LAlib.
#
LAdir        = $(HOME)/project/benchmarks/ATLAS/lib/Linux_HAMMER64SSE2_8
LAinc        =
LAlib        = $(LAdir)/libcblas.a $(LAdir)/libatlas.a
#
# ----------------------------------------------------------------------
# - F77 / C interface --------------------------------------------------
# ----------------------------------------------------------------------
# You can skip this section  if and only if  you are not planning to use
# a  BLAS  library featuring a Fortran 77 interface.  Otherwise,  it  is
# necessary  to  fill out the  F2CDEFS  variable  with  the  appropriate
# options.  **One and only one**  option should be chosen in **each** of
# the 3 following categories:
#
# 1) name space (How C calls a Fortran 77 routine)
#
# -DAdd_              : all lower case and a suffixed underscore  (Suns,
#                       Intel, ...),                           [default]
# -DNoChange          : all lower case (IBM RS6000),
# -DUpCase            : all upper case (Cray),
# -DAdd__             : the FORTRAN compiler in use is f2c.
#
# 2) C and Fortran 77 integer mapping
#
# -DF77_INTEGER=int   : Fortran 77 INTEGER is a C int,         [default]
# -DF77_INTEGER=long  : Fortran 77 INTEGER is a C long,
# -DF77_INTEGER=short : Fortran 77 INTEGER is a C short.
#
# 3) Fortran 77 string handling
#
# -DStringSunStyle    : The string address is passed at the string loca-
#                       tion on the stack, and the string length is then
#                       passed as  an  F77_INTEGER  after  all  explicit
#                       stack arguments,                       [default]
# -DStringStructPtr   : The address  of  a  structure  is  passed  by  a
#                       Fortran 77  string,  and the structure is of the
#                       form: struct {char *cp; F77_INTEGER len;},
# -DStringStructVal   : A structure is passed by value for each  Fortran
#                       77 string,  and  the  structure is  of the form:
#                       struct {char *cp; F77_INTEGER len;},
# -DStringCrayStyle   : Special option for  Cray  machines,  which  uses
#                       Cray  fcd  (fortran  character  descriptor)  for
#                       interoperation.
```

```
#
F2CDEFS      =
#
# ----------------------------------------------------------------------
# - HPL includes / libraries / specifics -------------------------------
# ----------------------------------------------------------------------
#
HPL_INCLUDES = -I$(INCdir) -I$(INCdir)/$(ARCH) $(LAinc) $(MPinc)
HPL_LIBS     = $(HPLlib) $(LAlib) $(MPlib)
#
# - Compile time options -----------------------------------------------
#
# -DHPL_COPY_L           force the copy of the panel L before bcast;
# -DHPL_CALL_CBLAS       call the cblas interface;
# -DHPL_CALL_VSIPL       call the vsip  library;
# -DHPL_DETAILED_TIMING  enable detailed timers;
#
# By default HPL will:
#    *) not copy L before broadcast,
#    *) call the BLAS Fortran 77 interface,
#    *) not display detailed timing information.
#
HPL_OPTS     = -DHPL_CALL_CBLAS
#
# ----------------------------------------------------------------------
#
HPL_DEFS     = $(F2CDEFS) $(HPL_OPTS) $(HPL_INCLUDES)
#
# ----------------------------------------------------------------------
# - Compilers / linkers - Optimization flags ---------------------------
# ----------------------------------------------------------------------
#
CC           = mpicc
CCNOOPT      = $(HPL_DEFS)
CCFLAGS      = $(HPL_DEFS) -fomit-frame-pointer -O3 -funroll-loops
#
# On some platforms,  it is necessary  to use the Fortran linker to find
# the Fortran internals used in the BLAS library.
#
LINKER       = mpif77
LINKFLAGS    = $(CCFLAGS)
#
ARCHIVER     = ar
ARFLAGS      = r
RANLIB       = echo
#
# ----------------------------------------------------------------------
```

Next, we ran the following command which began compilation:

Make arch=Hammer

The resulting HPL executable, *xhpl*, was placed in the HPL/bin directory.

Tuning

Within this project, we ran the HPL a total of 4 times. Since each run had a different number of nodes, and therefore a different total amount of available processing power and RAM, the HPL benchmark via HPL.dat had to be fine-tuned in order to obtain the best result possible given the available resources. If we picked a problem size too small, the HPL result would be far from its potential maximum. Similarly, if we chose a problem size too large, the job would either not finish, or performance would be dramatically degraded due to heavy swapping. What follows are our attempts at tuning HPL to obtain the best results possible for each scenario.

The first run was with 64 nodes (utilizing 512 cores). The nodes specified for this test run were just the compute nodes, not the master, and each compute node has 16GB of RAM. As such, this test run would utilize 1,048,576MB of RAM, or 2,048MB of RAM per core. Here is the resulting HPL.dat file that best matched this particular computing environment:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
331520         Ns
1            # of NBs
128           NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
16            Ps
32            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
##### This line (no. 32) is ignored (it serves as a separator). ######
```

```
0                               Number of additional problem sizes for PTRANS
1200 10000 30000                values of N
0                               number of additional blocking sizes for PTRANS
40 9 8 13 13 20 16 32 64        values of NB
```

The second run leveraged the full cluster (65 nodes). Since the master node only has 8GB of RAM, we totaled the RAM for each compute node (16,384MB * 64) with the RAM for the master node (8,192MB) which equaled 1,056,768MB. We then divided that total by 65 nodes to get an average RAM per node to use for the benchmark (16,258MB). What follows is the resulting HPL.dat that, we felt, best fits this particular scenario. Note that the Problem Size (Ns) is only slightly larger than the first HPL run as indicated above. Recall this when we explain the third HPL run.

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
332800         Ns
1            # of NBs
128           NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
20            Ps
26            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
##### This line (no. 32) is ignored (it serves as a separator). ######
```

```
0                               Number of additional problem sizes for PTRANS
1200 10000 30000                values of N
0                               number of additional blocking sizes for PTRANS
40 9 8 13 13 20 16 32 64        values of NB
```

The third run leveraged the full cluster again, but this time without reducing the total available RAM to compensate for the master node's 8GB. In this scenario, the total RAM that the HPL benchmark tried to use was 1,064,960MB. Due to this, swapping was heavily used on the master node, thereby decreasing HPL performance.

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
334080         Ns
1            # of NBs
128           NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
20            Ps
26            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
##### This line (no. 32) is ignored (it serves as a separator). ######
0                               Number of additional problem sizes for PTRANS
```

```
1200 10000 30000              values of N
0                             number of additional blocking sizes for PTRANS
40 9 8 13 13 20 16 32 64      values of NB
```

The fourth and final run again leveraged the full cluster and “pretended” that the master had double the RAM that it truly had, but this time the problem size was set very low, and the partition blocking factor was doubled.

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
1            # of problems sizes (N)
100000        Ns
1            # of NBs
256           NBs
0            PMAP process mapping (0=Row-,1=Column-major)
1            # of process grids (P x Q)
20            Ps
26            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
1            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
##### This line (no. 32) is ignored (it serves as a separator). ######
0                             Number of additional problem sizes for PTRANS
1200 10000 30000              values of N
0                             number of additional blocking sizes for PTRANS
40 9 8 13 13 20 16 32 64      values of NB
```

Each of the above scenarios relied upon the multitude of online data that explains how to fine-tune the HPL via it's HPL.dat file for optimal performance. The following is an excerpt directly from the online documentation for the HPL benchmark which explains the HPL.dat file in detail. [**14**]

Detailed Description of the HPL.dat File

Line 1: (unused) Typically one would use this line for its own good. For example, it could be used to summarize the content of the input file. By default this line reads:

```
     HPL Linpack benchmark input file
```

Line 2: (unused) same as line 1. By default this line reads:

```
Innovative Computing Laboratory, University of Tennessee
```

Line 3: the user can choose where the output should be redirected to. In the case of a file, a name is necessary, and this is the line where one wants to specify it. Only the first name on this line is significant. By default, the line reads:

HPL.out output file name (if any)

This means that if one chooses to redirect the output to a file, the file will be called "HPL.out". The rest of the line is unused, and this space to put some informative comment on the meaning of this line.

Line 4: This line specifies where the output should go. The line is formatted, it must begin with a positive integer, the rest is unsignificant. 3 choices are possible for the positive integer, 6 means that the output will go the standard output, 7 means that the output will go to the standard error. Any other integer means that the output should be redirected to a file, which name has been specified in the line above. This line by default reads:

```
     6        device out (6=stdout,7=stderr,file)
```

which means that the output generated by the executable should be redirected to the standard output.

Line 5: This line specifies the number of problem sizes to be executed. This number should be less than or equal to 20. The first integer is significant, the rest is ignored. If the line reads:

```
     3        # of problems sizes (N)
```

this means that the user is willing to run 3 problem sizes that will be specified in the next line.

Line 6: This line specifies the problem sizes one wants to run. Assuming the line above started with 3, the 3 first positive integers are significant, the rest is ignored. For example:

```
     3000 6000 10000    Ns
```

means that one wants xhpl to run 3 (specified in line 5) problem sizes, namely 3000, 6000 and 10000.

Line 7: This line specifies the number of block sizes to be runned. This number should be less than or equal to 20. The first integer is significant, the rest is ignored. If the line reads:

```
5         # of NBs
```

this means that the user is willing to use 5 block sizes that will be specified in the next line.

Line 8: This line specifies the block sizes one wants to run. Assuming the line above started with 5, the 5 first positive integers are significant, the rest is ignored. For example:

```
80 100 120 140 160 NBs
```

means that one wants xhpl to use 5 (specified in line 7) block sizes, namely 80, 100, 120, 140 and 160.

Line 9: This line specifies how the MPI processes should be mapped onto the nodes of your platform. There are currently two possible mappings, namely row- and column-major. This feature is mainly useful when these nodes are themselves multi-processor computers. A row-major mapping is recommended.

Line 10: This line specifies the number of process grid to be runned. This number should be less than or equal to 20. The first integer is significant, the rest is ignored. If the line reads:

```
2         # of process grids (P x Q)
```

this means that you are willing to try 2 process grid sizes that will be specified in the next line.

Line 11-12: These two lines specify the number of process rows and columns of each grid you want to run on. Assuming the line above (10) started with 2, the 2 first positive integers of those two lines are significant, the rest is ignored. For example:

```
1 2           Ps
6 8           Qs
```

means that one wants to run xhpl on 2 process grids (line 10), namely 1-by-6 and 2-by-8. Note: In this example, it is required then to start xhpl on at least 16 nodes (max of Pi-by-Qi). The runs on the two grids will be consecutive. If one was starting xhpl on more than 16 nodes, say 52, only 6 would be used for the first grid (1x6) and then 16 (2x8) would be used for the second grid. The fact that you started the MPI job on 52 nodes, will not make HPL use all of them. In this example, only 16 would be used. If one wants to run xhpl with 52 processes one needs to specify a grid of 52 processes, for example the following lines would do the job:

```
4  2          Ps
13 8          Qs
```

Line 13: This line specifies the threshold to which the residuals should be compared with. The residuals should be or order 1, but are in practice slightly less than this, typically 0.001. This line is made of a real number, the rest is not significant. For example:

```
16.0          threshold
```

In practice, a value of 16.0 will cover most cases. For various reasons, it is possible that some of the residuals become slightly larger, say for example 35.6. xhpl will flag those runs as failed, however they can be considered as correct. A run should be considered as failed if the residual is a few order of magnitude bigger than 1 for example 10^6 or more. Note: if one was to specify a threshold of 0.0, all tests would be flagged as failed, even though the answer is likely to be correct. It is allowed to specify a negative value for this threshold, in which case the checks will be by-passed, no matter what the threshold value is, as soon as it is negative. This feature allows to save time when performing a lot of experiments, say for instance during the tuning phase. Example:

```
     -16.0        threshold
```

The remaining lines allow to specifies algorithmic features. xhpl will run all possible combinations of those for each problem size, block size, process grid combination. This is handy when one looks for an "optimal" set of parameters. To understand a little bit better, let say first a few words about the algorithm implemented in HPL. Basically this is a right-looking version with row-partial pivoting. The panel factorization is matrix-matrix operation based and recursive, dividing the panel into NDIV subpanels at each step. This part of the panel factorization is denoted below by "recursive panel fact. (RFACT)". The recursion stops when the current panel is made of less than or equal to NBMIN columns. At that point, xhpl uses a matrix-vector operation based factorization denoted below by "PFACTs". Classic recursion would then use NDIV=2, NBMIN=1. There are essentially 3 numerically equivalent LU factorization algorithm variants (left-looking, Crout and right-looking). In HPL, one can choose every one of those for the RFACT, as well as the PFACT. The following lines of HPL.dat allows you to set those parameters.
Lines 14-21: (Example 1)

```
     3       # of panel fact
     0 1 2   PFACTs (0=left, 1=Crout, 2=Right)
     4       # of recursive stopping criterium
     1 2 4 8 NBMINs (>= 1)
     3       # of panels in recursion
     2 3 4   NDIVs
     3       # of recursive panel fact.
     0 1 2   RFACTs (0=left, 1=Crout, 2=Right)
```

This example would try all variants of PFACT, 4 values for NBMIN, namely 1, 2, 4 and 8, 3 values for NDIV namely 2, 3 and 4, and all variants for RFACT.
Lines 14-21: (Example 2)

```
     2       # of panel fact
     2 0     PFACTs (0=left, 1=Crout, 2=Right)
     2       # of recursive stopping criterium
     4 8     NBMINs (>= 1)
     1       # of panels in recursion
     2       NDIVs
     1       # of recursive panel fact.
     2       RFACTs (0=left, 1=Crout, 2=Right)
```

This example would try 2 variants of PFACT namely right looking and left looking, 2 values for NBMIN, namely 4 and 8, 1 value for NDIV namely 2, and one variant for RFACT.

In the main loop of the algorithm, the current panel of column is broadcast in process rows using a virtual ring topology. HPL offers various choices and one most likely want to use the increasing ring modified encoded as

1. 3 and 4 are also good choices.

Lines 22-23: (Example 1)

```
1       # of broadcast
1       BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
```

This will cause HPL to broadcast the current panel using the increasing ring modified topology.

Lines 22-23: (Example 2)

```
2       # of broadcast
0 4     BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
```

This will cause HPL to broadcast the current panel using the increasing ring virtual topology and the long message algorithm.

Lines 24-25 allow to specify the look-ahead depth used by HPL. A depth of 0 means that the next panel is factorized after the update by the current panel is completely finished. A depth of 1 means that the next panel is immediately factorized after being updated. The update by the current panel is then finished. A depth of k means that the k next panels are factorized immediately after being updated. The update by the current panel is then finished. It turns out that a depth of 1 seems to give the best results, but may need a large problem size before one can see the performance gain. So use 1, if you do not know better, otherwise you may want to try 0. Look-ahead of depths 3 and larger will probably not give you better results.

Lines 24-25: (Example 1):

```
1       # of lookahead depth
1       DEPTHs (>=0)
```

This will cause HPL to use a look-ahead of depth 1.
Lines 24-25: (Example 2):

```
2       # of lookahead depth
0 1     DEPTHs (>=0)
```

This will cause HPL to use a look-ahead of depths 0 and 1.

Lines 26-27 allow to specify the swapping algorithm used by HPL for all tests. There are currently two swapping algorithms available, one based on "binary exchange" and the other one based on a "spread-roll" procedure (also called "long" below). For large problem sizes, this last one is likely to be more efficient. The user can also choose to mix both variants, that is "binary-exchange" for a number of columns less than a threshold value, and then the "spread-roll" algorithm. This threshold value is then specified on Line 27.
Lines 26-27: (Example 1):

```
1       SWAP (0=bin-exch,1=long,2=mix)
60      swapping threshold
```

This will cause HPL to use the "long" or "spread-roll" swapping algorithm. Note that a threshold is specified in

that example but not used by HPL.
Lines 26-27: (Example 2):

```
2       SWAP (0=bin-exch,1=long,2=mix)
60      swapping threshold
```

This will cause HPL to use the "long" or "spread-roll" swapping algorithm as soon as there is more than 60 columns in the row panel. Otherwise, the "binary-exchange" algorithm will be used instead.

Line 28 allows to specify whether the upper triangle of the panel of columns should be stored in no-transposed or transposed form. Example:

```
0            L1 in (0=transposed,1=no-transposed) form
```

Line 29 allows to specify whether the panel of rows U should be stored in no-transposed or transposed form. Example:

```
0            U  in (0=transposed,1=no-transposed) form
```

Line 30 enables / disables the equilibration phase. This option will not be used unless you selected 1 or 2 in Line 26. Example:

```
1            Equilibration (0=no,1=yes)
```

Line 31 allows to specify the alignment in memory for the memory space allocated by HPL. On modern machines, one probably wants to use 4, 8 or 16. This may result in a tiny amount of memory wasted. Example:

```
8       memory alignment in double (> 0)
```

Appendix VII – Top 500 List for June, 2010

What follows is an abbreviated Top 500 List for June 2010 which was obtained here: http://www.top500.org/lists/2010/06.

The Percentage of Peak column was added in manually for purposes of this paper, as was the final Percentage of Peak average at the end of the list.

Rank	Computer	Cores	RMax	RPeak	% of Peak
1	Cray XT5-HE Opteron Six Core 2.6 GHz	224162	1759000	2331000	75.46%
2	Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU	120640	1271000	2984300	42.59%
3	BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband	122400	1042000	1375780	75.74%
4	Cray XT5-HE Opteron Six Core 2.6 GHz	98928	831700	1028850	80.84%
5	Blue Gene/P Solution	294912	825500	1002700	82.33%
6	SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon Westmere 2.93 Ghz, Infiniband	81920	772700	973291	79.39%
7	NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband	71680	563100	1206190	46.68%
8	eServer Blue Gene Solution	212992	478200	596378	80.18%
9	Blue Gene/P Solution	163840	458611	557056	82.33%
10	Sun Blade x6275, Xeon X55xx 2.93 Ghz, Infiniband	42440	433500	497396	87.15%
11	SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband	62976	433200	579379	74.77%
12	Blue Gene/P Solution	147456	415700	501350	82.92%
13	T-Platforms T-Blade2, Xeon 5570 2.93 GHz, Infiniband QDR	35360	350100	414419	84.48%
14	Sun Constellation, NovaScale R422-E2, Intel Xeon X5570, 2.93 GHz, Sun M9/Mellanox QDR Infiniband/Partec Parastation	26304	274800	308283	89.14%
15	Sun Blade x6048, X6275, IB QDR M9 switch, Sun HPC stack Linux edition	26232	274800	307439	89.38%
16	Cray XT6m 12-Core 2.1 GHz	43660	274700	366744	74.90%
17	Cray XT4 QuadCore 2.3 GHz	38642	266300	355506	74.91%
18	SGI Altix ICE 8200EX, Xeon E5472 3.0/X5560 2.8 GHz	23040	237800	267878	88.77%
19	Mole-8.5 Cluster Xeon L5520 2.26 Ghz, nVidia Tesla, Infiniband	33120	207300	1138440	18.21%
20	Cray XT4 QuadCore 2.1 GHz	30976	205000	260200	78.79%
21	Cray XT3/XT4	38208	204200	284000	71.90%
22	BX900 Xeon X5570 2.93GHz , Infiniband QDR	17072	191400	200080	95.66%
23	Blue Gene/P Solution	65536	190900	222822	85.67%
24	Dawning 5000A, QC Opteron 1.9 Ghz, Infiniband, Windows HPC 2008	30720	180600	233472	77.35%
25	Cluster Platform 3000 BL2x220, L54xx 2.5 Ghz, Infiniband	24704	179634	247040	72.71%
…	………………………………………………………………………	…….	....	…….	…..
101	Power 575, p6 4.7 GHz, Infiniband	3584	52810	67379.2	78.38%
102	Cluster Platform 3000 DL165, Opteron 2.5 GHz, 10GigE	7944	52192.1	79440	65.70%
103	Sun Blade X6440, Opteron 2.5 Ghz, Infiniband QDR	6464	51880	64640	80.26%
104	Power 575, p6 4.7 GHz, Infiniband	3520	51863.3	66176	78.37%
105	Power 575, p6 4.7 GHz, Infiniband	3520	51863.3	66176	78.37%
106	Altix 4700 1.6 GHz	9216	51441	58982	87.21%
107	Hitachi SR16000 Model L2/121, Power6 4.7Ghz, Infiniband	3872	51210	72793.6	70.35%
108	xSeries x3650M2 Cluster, Xeon QC E55xx 2.53 Ghz, GigE	8960	51203.3	90675.2	56.47%
109	Power 575, p6 4.7 GHz, Infiniband	3456	50923.9	64972.8	78.38%
110	NEC HPC 140Rb-1 Cluster, Xeon X5560 2.8Ghz, Infiniband	5376	50790	60211.2	84.35%

111	Fujitsu Cluster HX600, Opteron Quad Core, 2.3 GHz, Infiniband	6656	50510	61235	82.49%
112	Intel Cluster, Xeon X5560 2.8 GHz, Xeon X5670 2.93 Ghz, Infiniband	4824	50370	56225.3	89.59%
113	Sun Blade x6048, Xeon X5560 2.93 Ghz, Infiniband QDR	4600	49590	53912	91.98%
114	Sun Blade x6275, Xeon X55xx 2.8 Ghz, Infiniband QDR	4600	49590	51520	96.25%
115	Power 575, p6 4.7 GHz, Infiniband	3328	48932.7	62566.4	78.21%
116	Cluster Platform 3000 BL460c, Xeon 54xx 3.0GHz, GigEthernet	7600	48135.4	91200	52.78%
117	Blue Gene/P Solution	16384	47725	55706	85.67%
118	Blue Gene/P Solution	16384	47725	55706	85.67%
119	Blue Gene/P Solution	16384	47725	55706	85.67%
120	Blue Gene/P Solution	16384	47725	55706	85.67%
121	T-Platforms T60, Intel Quadcore 3Ghz, Infiniband DDR	5000	47170	60000	78.62%
122	Cluster Platform 3000 DL140 Cluster, Xeon 53xx 2.33GHz Infiniband	6440	47030	60020.8	78.36%
123	PowerEdge 1955, 2.66 GHz, Infiniband	5848	46730	62220	75.10%
124	xSeries x3650M2 Cluster, Xeon QC E55xx 2.53 Ghz, GigE	8096	46415.3	81931.5	56.65%
125	BladeCenter HS21 Cluster, Xeon QC HT 2.5 GHz, IB, Windows HPC 2008/CentOS	5376	46040	53760	85.64%
301	PRIMERGY RX200S5 Cluster, Xeon X5570 2.93GHz, Infiniband QDR	2880	31180	33753	92.38%
302	xSeries x3650 Cluster Xeon QC GT 2.66 GHz, Infiniband	3184	31153.4	33954.2	91.75%
303	xSeries x3650 Cluster Xeon QC GT 2.66 GHz, Infiniband	3184	31153.4	33954.2	91.75%
304	Cluster Platform 3000 BL460c, Xeon 54xx 3.0GHz, GigEthernet	4864	31126	58368	53.33%
305	Cluster Platform 3000 BL2x220, L54xx 2.5 Ghz, GigE	5856	31112.2	58560	53.13%
306	Cluster Platform 3000 BL460c G1, Xeon L5420 2.5 GHz, GigE	5856	31112.2	58560	53.13%
307	BladeCenter HS22 Cluster, Xeon QC GT 2.53 GHz, GigEthernet	5384	31078.2	54486.1	57.04%
308	BladeCenter HS22 Cluster, Xeon QC GT 2.53 GHz, GigEthernet	5384	31078.2	54486.1	57.04%
309	BladeCenter HS22 Cluster, Xeon QC GT 2.53 GHz, GigEthernet	5376	31032	54405.1	57.04%
310	BladeCenter HS22 Cluster, Xeon QC GT 2.53 GHz, GigEthernet	5376	31032	54405.1	57.04%
311	BladeCenter HS22 Cluster, Xeon QC GT 2.53 GHz, GigEthernet	5376	31032	54405.1	57.04%
312	BladeCenter HS22 Cluster, Xeon QC GT 2.53 GHz, GigEthernet	5376	31032	54405.1	57.04%
313	Cluster Platform 3000 BL460c G1, Xeon L5420 2.5 GHz, GigE	5840	31029.1	58400	53.13%
314	Cluster Platform 3000 BL460c G6, Xeon X5570 2.93 GHz, GigE	4960	30988.6	58131.2	53.31%
315	Cluster Platform 3000 BL460c, Xeon 54xx 3.0GHz, GigEthernet	4832	30925	57984	53.33%
316	Cluster Platform 3000 BL460c, Xeon 54xx 3.0GHz, GigEthernet	4832	30925	57984	53.33%
317	Cluster Platform 3000 BL460c, Xeon 54xx 3.0GHz, GigEthernet	4832	30925	57984	53.33%
318	Cluster Platform 3000 BL460c, Xeon 54xx 3.0GHz, GigEthernet	4832	30925	57984	53.33%
319	iDataPlex, Xeon X55xx QC 2.8 GHz, Infiniband	3072	30901.9	34406.4	89.81%
320	iDataPlex, Xeon X55xx QC 2.8 GHz, Infiniband	3072	30901.9	34406.4	89.81%
321	Bullx B500 Blade system, 2.4Ghz Intel L5330, QDR Infiniband	3744	30890	35942	85.94%
322	Bullx B500 Blade system, 2.4Ghz Intel L5330, QDR Infiniband	3744	30890	35942	85.94%
323	Cluster Platform 3000 BL460c, Xeon 54xx 3.0GHz, GigEthernet	4768	30522.7	57216	53.35%
324	Cluster Platform 3000 BL460c G1, Xeon L5410 2.33 GHz, GigE	6144	30389.7	57262.1	53.07%
325	Cluster Platform 3000 BL460c G1, Xeon L5410 2.33 GHz, GigE	6144	30389.7	57262.1	53.07%
476	Cluster Platform 3000 BL460c, Xeon 54xx 3.0GHz, GigEthernet	3844	24692.9	46128	53.53%
477	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	8064	24670	96768	25.49%
478	BladeCenter HS21 Cluster, Xeon QC HT 2.66 GHz, GigEthernet	8176	24670	87221.6	28.28%
479	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	7200	24670	86400	28.55%
480	BladeCenter HS21 Cluster, Xeon QC HT 2.83 GHz, GigEthernet	6992	24670	79149.4	31.17%
481	BladeCenter HS21 Cluster, Xeon QC HT 2.83 GHz, GigEthernet	6992	24670	79149.4	31.17%

482	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	6368	24670	76416	32.28%
483	BladeCenter HS21 Cluster, Xeon QC HT 2.66 GHz, GigEthernet	6816	24670	72713.1	33.93%
484	BladeCenter HS21 Cluster, Xeon QC HT 2.66 GHz, GigEthernet	6816	24670	72713.1	33.93%
485	BladeCenter HS21 Cluster, Xeon QC HT 2.66 GHz, GigEthernet	6336	24670	67592.4	36.50%
486	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
487	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
488	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
489	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
490	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
491	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
492	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
493	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
494	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
495	BladeCenter HS21 Cluster, Xeon QC HT 3 GHz, GigEthernet	5272	24670	63264	39.00%
496	BladeCenter HS21 Cluster, Xeon QC HT 2.66 GHz, GigEthernet	5744	24670	61277	40.26%
497	xSeries x3450 Cluster Xeon quad core, 3.0 GHz, GigEthernet	5096	24670	61152	40.34%
498	xSeries x3650 Cluster Xeon QC HT 2.66 GHz, GigEthernet	5152	24670	54940.9	44.90%
499	xSeries x3650 Cluster Xeon QC HT 2.66 GHz, GigEthernet	5152	24670	54940.9	44.90%
500	BladeCenter HS21 Cluster, Xeon QC 2.66 GHz, GigEthernet	5136	24670	54790.8	45.03%
				AVERAGE *P*:	**67.14%**

Appendix VIII – Original Java Code

What follows is the original Java code for the primary algorithm that was to be parallelized:

```
/**
*
*        bestDistance.java
*
*        Application that computes the best bands that minimize the distance
*        between one vector and a group of vectors (i.e. between the ground truth
*        target and the sampled truth vectors
*
*@author Stefan A. Robila
*@version 1.0
*@date 4/05/2007
*/

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;
import java.io.StreamTokenizer;

public class bestDistance
{
  public static final int VECSIZE=147;
  public static final int NUMVECTORS=5;

  private static double [][] x;

  public static void computeBestDistance(String  st, String en)
  {
    int i,j;
          String current = st;
          double min_distance = 10000;
          String best_distance_bands="";
          double current_distance=0;

          while (en.compareTo(current)>=0) {
             int numbands=0;

       //compute the norms (used in the lower part of the distance)
                 for (j=0; j<VECSIZE; j++) {
                                        if (current.charAt(j)=='1')
                                           numbands++;
                 }
                // System.out.println(numbands);
```

```
 if (numbands >= 2)
 {
    double [] distances = new double [NUMVECTORS];
          double [] norms = new double [NUMVECTORS];

           //compute the norms (used in the lower part of the distance)
          for (i=0; i<NUMVECTORS; i++){
                  norms[i]=0;
                  for (j=0; j<VECSIZE; j++) {
                           if (current.charAt(j)=='1')
                              norms[i] += x[i][j]*x[i][j];
                  }
                  norms[i]=Math.sqrt(norms[i]);
    }

//compute the norms (used in the lower part of the distance)
         current_distance = 0;
         for (i=1; i<NUMVECTORS; i++){
                  distances[i]=0;
                  for (j=0; j<VECSIZE; j++) {
                           if (current.charAt(j)=='1')
                                   distances[i] += x[0][j]*x[i][j];
                  }
                  distances[i]=distances[i]/(norms[0]*norms[i]);
                  current_distance += Math.acos(distances[i]);
         }
         current_distance /= (NUMVECTORS-1);

         // see if this is a better band combination
         if (current_distance < min_distance){
                  min_distance = current_distance;
                  best_distance_bands = current;
         }
//System.out.println(current+" "+current_distance);
 }
         //get the next band combination
         char cdata [] = current.toCharArray();
         j = VECSIZE-1;
         boolean done = false;
         while (j>=0 && !done) {
          if (cdata[j] == '1'){
                  cdata [j] = '0';
                  j--;
          }
          else {
            cdata[j] = '1';
                  done = true;
          }
   }
```

```
                    current = new String(cdata);
          }
          System.out.println(min_distance+" "+best_distance_bands);
         }

  /**
  *
  * The main portion of the application
  *
  **/
  public static void main ( String[] args )
  {

    //The application is started by three arguments
          // filename and string interval
    if ( args.length == 3 )
    {
              String filename = args[0];
        String st=args[1];
        String en=args[2];


                   if (st.length() != VECSIZE || en.length() != VECSIZE){
                                  System.err.println("The size of the bit strings must be EXACTLY
"+VECSIZE+" "+ st.length()+" "+en.length());
                                  return;
                   }


                   //initialize the vectors

                   x = new double[NUMVECTORS][VECSIZE];



                   //test if the search interval is correct
        if (en.compareTo(st)>=0)
        {
                                  //Need to read vector data here
                                  try {
                            readfileinVectors(filename);
                                  } catch (IOException e) {
                                          System.err.println("Caught IOException: "
                        + e.getMessage()+": "+en);
                                  }
            computeBestDistance(st,en);
        }
        else
        {
            System.out.println( "First string must be less than the second!");
        }
    }
    else
    {
       System.out.println( "Usage: <class_name> <file_name> <min> <max>");
    }
  }

    static void readfileinVectors(String filename) throws IOException {
                     Reader r = new BufferedReader(new FileReader(filename));
            StreamTokenizer stok = new StreamTokenizer(r);
          stok.parseNumbers();
            double sum = 0;
            stok.nextToken();

                     int count=0;
                     int v = 1;
                     boolean done=false;
                     //read data
            while (stok.ttype != StreamTokenizer.TT_EOF && !done) {
            if (stok.ttype == StreamTokenizer.TT_NUMBER) {
                     x[v][count] = stok.nval;
                                     count=(count+1)%VECSIZE;
                                     if (count==0){
                                             v++;
                                             if (v >= NUMVECTORS)
                                                     done=true;
                                     }
                             }
                             else
                     System.out.println("Nonnumber: " + stok.sval);
            stok.nextToken();
            }
                     for (count = 0; count<VECSIZE; count++)
                     {
                        x[0][count]=0;
                             for (v=1;v<NUMVECTORS;v++)
                               x[0][count]+=x[v][count];
                             x[0][count]/=NUMVECTORS;
                     }
        }
}
```

What follows is the original Java code that generated the intervals to be sent to the above-referenced algorithm; this also had to be converted to C/C++:

```
/**
*        generateQsubDistance.java
*
*        Application that computes the best bands that minimize the distance
*        between one vector and a group of vectors (i.e. between the ground truth
*        target and the sampled truth vectors
*
*@author Stefan A. Robila
*@version 1.0
*@date 4/05/2007
*/
public class generateQsubDistance
{
  public static int vecsize=0;
  public static int jobs=0;
  public static String filename="Blah";

  //note that jobs is the power of two for jobs
  public static void generateJobs() {
         String tail_of_zeros="";
         int i=vecsize-jobs;
          while (i>0) {
                 tail_of_zeros+='0';
                 i--;
          }

          //create the intervals
          String start_all="";
          String end_all="";
          i=jobs;
          while (i>0) {
                 start_all+='0';
                 end_all+='1';
                 i--;
          }

          String cStart=start_all;
          while (!cStart.equals(end_all)) {
                 //get the next band combination
                 char cdata [] = cStart.toCharArray();
                 int j = cStart.length()-1;
                 boolean done = false;
                 while (j>=0 && !done) {
                          if (cdata[j] == '1'){
                                 cdata [j] = '0';
                                 j--;
                     }
```

```
                          else {
                            cdata[j] = '1';
                                 done = true;
                          }
                  }
                  String cEnd = new String(cdata);

                  System.out.println("qsub -r y java bestDistance "+filename+"
"+(cStart+tail_of_zeros)+" "+(cEnd+tail_of_zeros));
                  cStart = cEnd;
          }
}
  /**
  *
  * The main portion of the application
  *
  **/
  public static void main ( String[] args )
  {
    //The application is started by three arguments
          // filename and string interval
    if ( args.length == 3 )
    {
            filename = args[0];
            try {
             vecsize = Integer.parseInt(args[1]);
             jobs = Integer.parseInt(args[2]);
                  } catch (NumberFormatException e) {
                         System.err.println("Caught NumberException: "
                       + e.getMessage()+": not an integer number");
                                 return;
                  }

                   System.out.println("Size: "+vecsize+", jobs: "+jobs);
                   if (vecsize < jobs) {
                      System.out.println("Incorrect use, num bands smaller than power num
                                             intervals");
                              return;
                   }
                   generateJobs();
    }
    else
    {
       System.out.println( "Usage: <class_name> <file_name> (147) <num_bands> <num_jobs>");
    }
  }
}
```

Appendix VIIII – NAS Parallel Benchmark Suite; Technical Details

The *make.def* file used for compiling the benchmarks we used during this project:

```
#---------------------------------------------------------------------------
#
#                SITE- AND/OR PLATFORM-SPECIFIC DEFINITIONS.
#
#---------------------------------------------------------------------------

#---------------------------------------------------------------------------
# Items in this file will need to be changed for each platform.
# (Note these definitions are inconsistent with NPB2.1.)
#---------------------------------------------------------------------------

#---------------------------------------------------------------------------
# Parallel Fortran:
#
# For CG, EP, FT, MG, LU, SP and BT, which are in Fortran, the following must
# be defined:
#
# MPIF77     - Fortran compiler
# FFLAGS     - Fortran compilation arguments
# FMPI_INC   - any -I arguments required for compiling MPI/Fortran
# FLINK      - Fortran linker
# FLINKFLAGS - Fortran linker arguments
# FMPI_LIB   - any -L and -l arguments required for linking MPI/Fortran
#
# compilations are done with $(MPIF77) $(FMPI_INC) $(FFLAGS) or
#                            $(MPIF77) $(FFLAGS)
# linking is done with       $(FLINK) $(FMPI_LIB) $(FLINKFLAGS)
#---------------------------------------------------------------------------

#---------------------------------------------------------------------------
# This is the fortran compiler used for MPI programs
#---------------------------------------------------------------------------
MPIF77 = mpif77
# This links MPI fortran programs; usually the same as ${MPIF77}
FLINK   = $(MPIF77)

#---------------------------------------------------------------------------
# These macros are passed to the linker to help link with MPI correctly
#---------------------------------------------------------------------------
```

```
FMPI_LIB  =

#---------------------------------------------------------------------------
# These macros are passed to the compiler to help find 'mpif.h'
#---------------------------------------------------------------------------
FMPI_INC =

#---------------------------------------------------------------------------
# Global *compile time* flags for Fortran programs
#---------------------------------------------------------------------------
FFLAGS  = -O3
# FFLAGS = -g

#---------------------------------------------------------------------------
# Global *link time* flags. Flags for increasing maximum executable
# size usually go here.
#---------------------------------------------------------------------------
FLINKFLAGS = -O3


#---------------------------------------------------------------------------
# Parallel C:
#
# For IS, which is in C, the following must be defined:
#
# MPICC      - C compiler
# CFLAGS     - C compilation arguments
# CMPI_INC   - any -I arguments required for compiling MPI/C
# CLINK      - C linker
# CLINKFLAGS - C linker flags
# CMPI_LIB   - any -L and -l arguments required for linking MPI/C
#
# compilations are done with $(MPICC) $(CMPI_INC) $(CFLAGS) or
#                            $(MPICC) $(CFLAGS)
# linking is done with       $(CLINK) $(CMPI_LIB) $(CLINKFLAGS)
#---------------------------------------------------------------------------

#---------------------------------------------------------------------------
# This is the C compiler used for MPI programs
#---------------------------------------------------------------------------
MPICC = mpicc
# This links MPI C programs; usually the same as ${MPICC}
CLINK   = $(MPICC)
```

```
#---------------------------------------------------------------------------
# These macros are passed to the linker to help link with MPI correctly
#---------------------------------------------------------------------------
CMPI_LIB  =

#---------------------------------------------------------------------------
# These macros are passed to the compiler to help find 'mpi.h'
#---------------------------------------------------------------------------
CMPI_INC =

#---------------------------------------------------------------------------
# Global *compile time* flags for C programs
#---------------------------------------------------------------------------
CFLAGS  = -O3
# CFLAGS = -g

#---------------------------------------------------------------------------
# Global *link time* flags. Flags for increasing maximum executable
# size usually go here.
#---------------------------------------------------------------------------
CLINKFLAGS = -O3


#---------------------------------------------------------------------------
# MPI dummy library:
#
# Uncomment if you want to use the MPI dummy library supplied by NAS instead
# of the true message-passing library. The include file redefines several of
# the above macros. It also invokes make in subdirectory MPI_dummy. Make
# sure that no spaces or tabs precede include.
#---------------------------------------------------------------------------
# include ../config/make.dummy


#---------------------------------------------------------------------------
# Utilities C:
#
# This is the C compiler used to compile C utilities.  Flags required by
# this compiler go here also; typically there are few flags required; hence
# there are no separate macros provided for such flags.
#---------------------------------------------------------------------------
CC      = mpicc -g
```

```
#---------------------------------------------------------------------------
# Destination of executables, relative to subdirs of the main directory. .
#---------------------------------------------------------------------------
BINDIR  = ../bin


#---------------------------------------------------------------------------
# Some machines (e.g. Crays) have 128-bit DOUBLE PRECISION numbers, which
# is twice the precision required for the NPB suite. A compiler flag
# (e.g. -dp) can usually be used to change DOUBLE PRECISION variables to
# 64 bits, but the MPI library may continue to send 128 bits. Short of
# recompiling MPI, the solution is to use MPI_REAL to send these 64-bit
# numbers, and MPI_COMPLEX to send their complex counterparts. Uncomment
# the following line to enable this substitution.
#
# NOTE: IF THE I/O BENCHMARK IS BEING BUILT, WE USE CONVERTFLAG TO
#       SPECIFIY THE FORTRAN RECORD LENGTH UNIT. IT IS A SYSTEM-SPECIFIC
#       VALUE (USUALLY 1 OR 4). UNCOMMENT THE SECOND LINE AND SUBSTITUTE
#       THE CORRECT VALUE FOR "length".
#       IF BOTH 128-BIT DOUBLE PRECISION NUMBERS AND I/O ARE TO BE ENABLED,
#       UNCOMMENT THE THIRD LINE AND SUBSTITUTE THE CORRECT VALUE FOR
#       "length"
#---------------------------------------------------------------------------
# CONVERTFLAG   = -DCONVERTDOUBLE
# CONVERTFLAG   = -DFORTRAN_REC_SIZE=length
# CONVERTFLAG   = -DCONVERTDOUBLE -DFORTRAN_REC_SIZE=length


#---------------------------------------------------------------------------
# The variable RAND controls which random number generator
# is used. It is described in detail in Doc/README.install.
# Use "randi8" unless there is a reason to use another one.
# Other allowed values are "randi8_safe", "randdp" and "randdpvec"
#---------------------------------------------------------------------------
RAND   = randi8
# The following is highly reliable but may be slow:
# RAND   = randdp
```

The *suite*.def file used for compiling the benchmarks we used during this project:

```
# config/suite.def
# This file is used to build several benchmarks with a single command.
# Typing "make suite" in the main directory will build all the benchmarks
# specified in this file.
# Each line of this file contains a benchmark name, class, and number
# of nodes. The name is one of "cg", "is", "ep", mg", "ft", "sp", "bt",
# "lu", and "dt".
# The class is one of "S", "W", "A", "B", "C", and "D".
# The number of nodes must be a legal number for a particular
# benchmark. The utility which parses this file is primitive, so
# formatting is inflexible. Separate name/class/number by tabs.
# Comments start with "#" as the first character on a line.
cg	A	512
cg	B	512
cg	C	512
cg	D	512
is	A	512
is	B	512
is	C	512
ep	A	512
ep	B	1
ep	B	2
ep	B	4
ep	B	8
ep	B	16
ep	B	32
ep	B	64
ep	B	128
ep	B	256
ep	B	512
ep	B	520
ep	C	512
ep	D	512
lu	A	1
lu	A	2
lu	A	4
lu	A	8
lu	A	16
lu	A	32
lu	A	64
lu	A	128
lu	A	256
lu	B	1
```

```
lu      B       2
lu      B       4
lu      B       8
lu      B       16
lu      B       32
lu      B       64
lu      B       128
lu      B       256
lu      C       32
lu      C       64
lu      C       128
lu      C       256
lu      D       64
lu      D       128
lu      D       256
lu      D       512
```