High Performance Computing Resilience Testbed

Hector J. Machin
Research Alliance in Math and Science
University of Turabo, Puerto Rico

August 2009

Thomas Naughton, Geoffroy Vallée, Stephen L. Scott and Anand Tikotekar
Computater Science and Mathematic Division
Oak Ridge National Laboratory

Prepared for
(Office of Science, U.S. Department of Energy)
Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6285
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725
Table of Contents

I. Introduction ......................................................................................................................... 3

II. Background .......................................................................................................................... 4

III. Overview .............................................................................................................................. 4

IV. Resources ............................................................................................................................ 5-9

V. Methodology ......................................................................................................................... 9-11

VI. Experiments .......................................................................................................................... 11-14

VII. Results ................................................................................................................................. 14-15

VIII. Conclusions ......................................................................................................................... 16

IX. Future work .......................................................................................................................... 16
Abstract

Scientific computing applications are becoming increasingly complex. Clusters and parallel systems then become an important factor for the execution of these applications. The probability of failures increases as the complexity and scale of these systems increases. In this research, the effects of generating representative faults (resilience testbed) were explored, these can be useful when developing methods for fault correction and failure prevention. Fault injection techniques used while executing application benchmarks to provide a controlled environment to study simulated failures on an actual running application. Through the development of a resilience testbed for high performance computing (HPC) the following was investigated: fault injection mechanisms, streamlining of data gathering/monitoring, data analyzers for failure evaluation, and the validation of failure detection techniques. The system used for this research was the Oak Ridge National Laboratory, XTORC cluster. The HPCC parallel benchmark was used during the experiments. Perl scripts were developed to automate the processes of: (i) data gathering/monitoring startup, (ii) benchmark execution, (iii) fault injection, (iv) data gathering/monitoring shutdown, (iv) data archival, and (v) event logging for the experiment. The multivariate data has been projected and analyzed to properly study the system using GGobi, an open source visualization program for exploring high-dimensional data, and R, a software environment for statistical computing and graphics. Then from the analyzed data, a hypothesis was developed for failure detection. Straight comparison between the hypothesis and the experiment event log provides a basis for validation of the given analyzers. This research can provide a better understanding of which monitoring metrics (datasets) for a supercomputer have a direct relationship with node failure. This knowledge can aid in the development of systems for real time error tracking and failure management, which are important in order to improve HPC system efficiency and application resilience.

Index Terms—Fault Tolerance, HPC (High Performance Computing), MPI (Message Passing Interface), HPCC Parallel Benchmarks (High Performance Computing Challenge Parallel Benchmarks).

1 INTRODUCTION

Fault tolerance in high performance computing (HPC) systems has been researched for many years. As supercomputing power reaches petascale computing power, preventing failure of the application is a more evident problem. The most widely used fault tolerance method now days are checkpoints. Checkpoints are stopping the running application and writing images of its state into memory. If the application fails, it would restart with the last checkpoint made. This fault tolerance method is projected to become useless soon. As the cores per chip keeps doubling each thirty months; the supercomputer system utilization will drop to zero by year 2013. This would be caused because the system will spend 100% of its useful time writing checkpoints to memory [1]. For these reasons, grows a need to search for other ways to prevent application failures. Through this paper we are going to develop a resilience testbed for automating steps, using Perl scripts, of starting the experimental application, fault injection, data
gathering and monitoring and keeping a log of every event of each experiment realized.

The remainder of this paper is organized as follows. Section 2 gives a background for this research. Section 3 describes an overview of this testbed. Section 4 depicts all the resources: open source applications used and the targeted system. Section 5 shows in detail the methodology used. Section 6 describes experiments realized. Section 7 discusses results from experiments in the previous section. Section 8 gives the conclusions for the research and future work is given on Section 9.

2 Background
One of the most challenging problems that will desperately need to be overcome by HPC systems will be avoiding, coping and recovering from failures. The current and coming PetaFLOPS systems require the use and control of hundreds of thousands or even millions of processing, storage, and networking elements. This great number of elements will increase the probability of facing failures, degrading the application performance on these systems [1]. The progress of petascale computing will rely on the accuracy of fault tolerance achieved by the time.

In the area of fault tolerance, fault injection makes one of the most important topics. Fault injection is the purposeful introduction of faults (or errors) into a target [2]. Faults can be injected by either software or hardware [3]. On comparison the hardware injected faults requires specialized equipment, and may be hazardous for the target system in use. In the other hand software injected faults are harmless to the hardware and can replicate the effects of the hardware faults into the system and the testing application. In a software based approach, or software implemented fault injection (SWIFI), there is more flexibility in terms of how to implement and detect the faults. However, the faults are limited in scope to those that are accessible via software [2]. Through this research we concentrate on building a resilience testbed to introduce software-based errors.

3 Overview
As we mentioned before our resilience testbed is composed of five main elements:
1. Fault injection techniques
2. Data gathering/monitoring
3. Testing application
4. Target System
5. Analyzers for failure evaluation

These are represented on figure 1. Fault injection techniques were explored with the purpose of injecting them into the running application and/or to the target system. The target system would be monitored while a testing application is being executed. The data gathered from the monitoring would be archived and send to an analyzer for exploring if these analyzer being developed could detect the abnormal behavior of the system in presence of faults, proximity to failure and source of these faults injected. In the following section a detailed description of the resources used for each of the ele-
ments will be provided.

4 RESOURCES
4.1 Fault Injection Techniques
The fault injection techniques explored were:

1. Register bit flips
2. Network congestion
3. Killing an instance of the application on one node
4. Overloading

Through these injected faults we could begin to appreciate abnormalities of the nodes of the system when faults are being injected.

4.1.1 Register bit flips
 Registers are the inner processor memory. Absolutely everything (variables, dataflow, etc.) during the operation of a processor will be received by the registers and sent to the processor for calculations. Target bit flips in the EAX (Accumulator Register) register during the execution of an application will alter the flow of the program ending, hopefully, in the crash of the application. We explored the fault injections made by ptrace linux system call tipe FILE*, e.g., fopen() [1].

4.1.2 Network congestion
 Network congestion was explored as a fault injection technique. The idea behind is to generate a congestion that would stress the target system, limiting it from normal execution. Hopefully this stress would prove to be so strong that would end in application termination. This would prove to be the most efficient fault injection method of the ones explored in this premature testbed (explained in the subsequent sections). The type of network congestion injected is the unix command ping.

4.1.3 Killing an instance of the application on one node
Killing an instance of the application on one node is an “effective” way for instantly depriving a node from workload. This serves completely for our final purposes, but fails in the intermediate steps (see results section).
4.1.4 Overloading
By overloading, is meant to indicate the process of running several instances of the same application. Also this method proves to be efficient for final purposes but dangerous in some aspects (see results section).

4.2 Data gathering/monitoring
Data gathering and monitoring was able by two open source resources: Ganglia and Ovis 1.1

4.2.1 Ganglia
Ganglia is a scalable distributed system monitor tool for high-performance computing systems such as clusters and grids. It allows the user to remotely view live or historical statistics (such as CPU load averages or network utilization) for all machines that are being monitored. Ganglia is based on a hierarchical design targeted at federations of clusters. It relies on a multicast-based listen/announce protocol to monitor state within clusters and uses a tree of point-to-point connections amongst representative cluster nodes to federate clusters and aggregate their state. It leverages widely used technologies such as XML for data representation, XDR for compact, portable data transport, and RRDtool for data storage and visualization. It uses carefully engineered data structures and algorithms to achieve very low per-node overheads and high concurrency.

The implementation is robust, has been ported to an extensive set of operating systems and processor architectures, and is currently in use on over 500 clusters around the world. It has been used to link clusters across university campuses and around the world and can scale to handle clusters with 2000 nodes [4]. Ganglia has four important components [5]:
1. gmond
2. gmetad
3. gmetric
4. gstat
5. web

4.2.1.1 gmond
The ganglia monitoring daemon (gmond) is a lightweight service that is installed on every machine monitored. This daemon uses a simple listen/announce protocol via XDR to collect monitoring state and then shares this information via XML over TCP. Gmond is portable and collects dozens of system metrics: CPU, memory, disk, network and process data.

4.2.1.2 gmetad
The ganglia meta daemon (gmetad) is a service that collects data from other gmetad and gmond sources and stores their state to disk in indexed round-robin databases. Gmetad provides a simple query mechanism for collecting historical information about groups of machines. Gmetad supports hierarchical delegation for creating manageable monitoring domains.
4.2.1.3 gmetric
The ganglia metric tool is a commandline application that can be used to inject custom made metrics about hosts that are being monitored by ganglia. It has the ability to spoof messages as coming from a different host in case a user wants to capture and report metrics from a device where gmond is not running (like a network or other embedded device).

4.2.1.4 gstat
The ganglia stat tool is a commandline application that can be used to query a gmond directly.

4.2.1.5 web
The ganglia web frontend expresses the data stored by gmetad in a graphical web interface using PHP.

4.2.2 Ovis 1.1
Ovis is a tool for intelligent analysis and monitoring of large computational clusters. Through an intuitive user-friendly GUI, OVIS provides 2D visualization capabilities of cluster health parameters as well as a number of statistical tools. This allows at-a-glance detection of abnormalities in a system or node depending on how the user has tuned the interface. Furthermore, the correlation engine provided with the distribution allows the user to analyze statistical relationships between different parameters of the system. Knowledge of these relationships can then greatly reduce the parameter space that the user needs to actively monitor in order to detect abnormal node behavior. This is in contrast to current cluster monitoring methodologies in which data is obtained from each node and a pre-defined rule set is applied, on a per-node basis, to any node whose value(s) cross a pre-determined threshold(s). Though this methodology is well suited to single nodes and small clusters, OVIS uses the statistical properties of these large collections of statistically similar devices to add a great deal of intelligence to the process of monitoring and analysis as well as being able to determine many problems sooner than is possible using static thresholds [6].

4.2.2.1 Ovis 1.1 data gathering
The ovis_RRD_XXX scripts work together to read from RRDs such as those populated by Ganglia, via the RRDtool perl interface. They do not interact with Ganglia itself. The assumption is that there exists a main directory containing subdirectories, one for each compute node. Within those subdirectories are the RRDs for that node, one for each different variable to be monitored. Each RRD could have multiple timescales being considered (e.g., for data taken on 5 second intervals, the RRD may contain: 1 single datapoint AVERAGE, 2 minute AVERAGE, 10 minute AVERAGE, etc.), but it must correspond to one quantity only (e.g., cpu1 temp). The rrd names are assumed to be <variable>.rrd, e.g., cpu1 temp.rrd [6].
4.3 Testing application
The application in use for testing purposes during this testbed are the HPCC parallel benchmarks. Also the open source application Cbench is used for nicely running and analyzing output of these benchmarks.

4.3.1 HPCC parallel benchmarks
The HPCChallenge suite of benchmarks will examine the performance of HPC architectures using kernels with memory access patterns more challenging than those of the High Performance Linpack (HPL) benchmark used in the Top500 list. The HPCChallenge suite is being designed to augment the Top500 list, provide benchmarks that bound the performance of many real applications as a function of memory access characteristics e.g., spatial and temporal locality, and provide a framework for including additional benchmarks. The HPCChallenge benchmarks are scalable with the size of data sets being a function of the largest HPL matrix for a system. The HPCChallenge benchmark suite has been released by the DARPA HPCS program to help define the performance boundaries of future Petascale computing systems. The suite is composed of several well known computational kernels (STREAM, High Performance Linpack, matrix multiply – DGEMM, matrix transpose, FFT, RandomAccess, and bandwidth/latency tests) that attempt to span high and low spatial and temporal locality space [7].

4.3.1 Cbench
Cbench [11] is:
- a Perl-based scripting framework for building, running, and analyzing the output of various opensource codes
- a highly useful toolkit for stressing a system for maintenance or acceptance testing
- an easy way to benchmark and analyze a cluster using any of a variety of tests
- a project created by HPC system administrators and engineers at Sandia National Labs
- as a toolkit, there are many ways that Cbench can be utilized for many different Linux cluster testing tasks

4.4 Target system
For the target system, Oak Ridge National Laboratory, Computer Science and Mathematics Division XTORC cluster was used.

4.4.1 XTORC cluster
Table 1. System specifications

<table>
<thead>
<tr>
<th>System name</th>
<th>XTORC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes</td>
<td>64</td>
</tr>
<tr>
<td>Processors per node</td>
<td>1</td>
</tr>
<tr>
<td>CPU type</td>
<td>Pentium VI</td>
</tr>
<tr>
<td>CPU speed</td>
<td>1695.7MHz</td>
</tr>
<tr>
<td>Memory per node</td>
<td>768MB</td>
</tr>
</tbody>
</table>
4.5 Analyzers for failure evaluation

The analyzers are outside of the scope of this research. This research results focused on archiving gathering monitored data from each fault injection experiment done in the target system. Results where passed on to a set of collaborators [8] where they explored clustering techniques and visualization tools using the open source applications: R and Ggobi.

4.5.1 R

R is an integrated GNU project suite of software facilities for data manipulation, calculation and graphical display [9]. It includes:

- An effective data handling and storage facility
- A suite of operators for calculations on arrays, in particular matrices
- A large, coherent, integrated collection of intermediate tools for data analysis
- Graphical facilities for data analysis and display either on-screen or on hardcopy
- A well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities.

4.5.2 Ggobi

Ggobi is an open source visualization program for exploring high-dimensional data. It provides highly dynamic and interactive graphics such as tours, as well as familiar graphics such as the scatterplot, barchart and parallel coordinates plots. Plots are interactive and linked with brushing and identification [10].

5 Methodology

The organization of the system being developed has the following components (see figure 3):

1. Event log
2. RRD starter  
   a. RRD reader  
3. Start the application  
4. Wait until the application finishes while injecting faults  
5. RRD killer  
6. Data archival  

![Fig. 3. Research methodology](image)

### 5.1 Event log
For keeping an accurate record of what happens in each experiment an event log is incorporated to the system. This event log will keep track of: start of the application, type and time of fault injected, and end of the application. The event log will later be used to prove or disprove the predictions made by the analyzers for failure evaluation. Also this log is useful for understanding the results.

### 5.2 RRD starter
The ovis RRD launcher launches readers, one for each subdirectory of the main directory where the RRD databases are kept, these correspond to compute nodes. Coordinates the start times of all the ovis RRD readers if a start time is not specified. It can use a predefined header file, as mentioned, with respect to the ovis RRD reader or it will create a header that is the union of all the subdirs rrd names [6].

#### 5.2.1 RRD reader
The ovis RRD reader will read from the RRDS for a single node. It will extract info from the RRDS for that node, and determine for each RRD, what the minimum measurement interval is (e.g., in the example above it will discern that the 1 datapoint average is the shortest interval (taken on 5 second intervals). It will then compare across all the RRDS and only collect data from those whose shortest interval corresponds to the global shortest interval. For example, if 5 seconds is the shortest interval across all RRDS, then data will only be collected from RRDs who have a quantity whose shortest interval is 5 seconds [6].

These readers produce the data files, and optionally the header file. If the header file is being produced by the ovis RRD reader then it will check all possible RRDS for inclusion in the set. If an existing headerfile is specified on the command line, then only those
quantities will be considered for inclusion in the set [6].

5.3 Start of the application
Two steps done by Cbench start the application. First the gen_jobs script creates a specific job, for a specific benchmark, problem size and number of processors. After this job has been created, the start_job scripts would let us nicely start our benchmark interactively or using batch system.

5.4 Wait until the application finishes while injecting faults
We divide this segment in two parts: inject faults while application is still running and after the application finishes check for failure of the application.

5.4.1 Inject faults
The faults injected are listed in section 4.1. The tools used primarily for injecting these faults are the cluster command control (C3) [12] tools. These tools will let me run from head node any kind of executable in any node desired. Also they would let me kill an instance of the application on any node desired.

5.4.2 Check for failure
The check of the failure of the system is done efficiently by using the Cbench script output_parse. This script, besides letting you see the results of the running application and generate performance plots in gnuplot, describes at what percent did the benchmark executed. With this output we can be certain of if the application really failed or just ended.

5.5 RRD killer
The ovis RRD killer script will kill all processes with "ovis RRD reader" in the name for ease of killing every instance of this process [6]. Also for the process of killing the RRD reader, a delaying time can be specified in the input file to run the experiments. This can nicely shutdown the data monitoring process on the target system.

5.6 Data archival
The data archival process consists on gathering all the data generated for the specific experiments and build tarballs of these data with the experiment name.

6 Experiments
For the experiments we developed an input file which you specify certain parameters of the experiment at use.

```
[ experiment_name ]
    workload =
    number_nodes =
    mpinodes_inputfile =
    delaytime =
    faultinjection =
```

Fig. 4. Experiment input file parameters
The parameters are listed as:

1. `experiment_name` - name of the experiment always has to be enclosed in brackets.
2. `workload` - type of workload being run by the application, e.g. benchmark at hand. If you are only running an unix command you can specify it too in this parameter (ex. `workload = sleep 2`).
3. `number_nodes` - number of nodes that you wish to execute your workload.
4. `mpinodes_inputfile` - this is the input file path of the name of the nodes or hosts that are going to be used for run the experiment. Also known as the lamboot input file, or the file where the name of the host from which the MPI environment will be set up by lam.
5. `delaytime` - delay time from the start/stop of the data gathering to the start/stop of the application. Given in seconds.
6. `faultinjection` - specify the path to the executable of the given fault, or execute any custom made command.

Also there is a tool from my system to specify any number of experiments in this input file. See figure 5. A custom option was made for the option of running only one of the experiments or to run them all in order from the first in top of the input file to the last in

```
[ sleep2hrs ]
  workload = sleep 13200
  number_nodes = n
  mpinodes_inputfile =
  delaytime = 0
  faultinjection =

[ hpc_one_node ]
  workload = hpc
  number_nodes = 1
  mpinodes_inputfile = nodes.xtorc
  delaytime = 300
  faultinjection =

[ hpc_all_nodes ]
  workload = hpc
  number_nodes = 32
  mpinodes_inputfile = nodes.xtorc
  delaytime = 300
  faultinjection =

[ kill_fault ]
  workload = hpc
  number_nodes = 32
  mpinodes_inputfile = nodes.xtorc
  delaytime = 300
  faultinjection = csc:10 hpc

[ network_fault ]
  workload = hpc
  number_nodes = 32
  mpinodes_inputfile = nodes.xtorc
  delaytime = 300
  faultinjection = csc:10,20,32,40 ./infiniteping.pl

[ overload ]
  workload = hpc
  number_nodes = 32
  mpinodes_inputfile = nodes.xtorc
  delaytime = 300
  faultinjection = csc: hpc
```

Fig. 5. All experiments in the input file
the bottom of the input file. At last the figure 6 shows us the options to run the script developed to run all this system the experiments.pl perl script

```
Usage: ./experiments.pl <inputfilepath> [options]
Options: 
  -run=all run all experiments specified in <filepath>
  -run=experiment_name specify which experiment you want to run only
  -genplots generates plots of all nodes and metrics at the
    end of each experiment
```

Fig. 6. Usage of the experiments.pl script

In the following table 2 we can see the list of experiments mad

Table 2. Experiments list.

<table>
<thead>
<tr>
<th>Experiment name</th>
<th>Workload</th>
<th>Fault injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>sleep12hrs</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>hpcc_onenode</td>
<td>hpcc</td>
<td>none</td>
</tr>
<tr>
<td>hpcc_allnodes</td>
<td>hpcc</td>
<td>none</td>
</tr>
<tr>
<td>kill_fault</td>
<td>hpcc</td>
<td>kill instance app</td>
</tr>
<tr>
<td>network_fault</td>
<td>hpcc</td>
<td>infinite ping</td>
</tr>
<tr>
<td>overload</td>
<td>hpcc</td>
<td>several instances</td>
</tr>
</tbody>
</table>

The purpose of first three experiments mentioned on table 2 was to test the system developed and provide initial and control groups of data sets to the collaborators [8] working with the analyzers for failure evaluations.

The kill_fault was described in section 4.1.3. In that moment we introduced as a success on our final purposes that is to make the application fail. But we also mention that this method of fault injection fails on intermediate steps. In a more detailed reasons, we should analyze two cases: 1) running a parallel application in the target system, 2) running several instances of a serial application in the target system. For case one this method fails in showing us how the system would behave in the proximity of encountering failures. It also would fail in representing which nodes behave abnormally as the fault injected nodes. All these reasons try to approach the first case being a message passing interface (MPI) parallelized application. If an instance of this application is killed on one of the nodes of the targeted system the whole application will fail instantly. Creating a whole system application failure without none or minimum trace of which of the nodes where fault injected. For the second case this method is fully successful, but for as we moved on this research we became more interested on fully parallelized applications (case 1) and discarded this experiment.

The network_fault is the most useful experiment we developed in a short period of time. The network_fault consisted on running infinite pings on several nodes, making the system stress and the application fails eventually leaving an interesting trace in its monitoring/gathered data. The results of this experiment will be shown in the next sec-
The overload experiment is based on section 4.1.4 overloading injection fault. During this experiment we ran several instances of the hpcc parallel benchmark on the system. As said in section 4.1.4 is an efficient but dangerous method. This statement is based on actual results. The results of these experiments are a chain of systems failures that caused in a few seconds the failure of fifteen nodes. For these reasons we took precautions for a shorter time term to not re-run this experiment again.

7 Results
In this section we are going to describe the results of the experiment network_fault. Through the following figures 7, 8 and 9 we can see a report in which each sin the plot square represents a different node. The node is identified by the number in top of the square. in all the figures the squares that just show a straight blue line and no sight of movement is because those nodes where not used in that experiment or more likely they where dead at the time of the experiment. We encounter three different behaviors: 1) the node quickly finished the workload, 2) the node appears to be in pressure of workload through all the experiment and 3) the node starts doing normal workload and at some point increases very quickly the workload on it.
For the first case, appears to be that the hpcc benchmarks does not divides the tasks equally among all the nodes. For the second case, in my opinion these nodes could not finish their workload because they where waiting on sending and receiving messages from the nodes in the third case. In the third case, which can appear obvious, are the fault injected nodes. The ones that are marked as blue (nodes 10, 20, 32 and 40) are the first ones that the infinite ping fault was injected. In time they gather a lot of network congestion and workload on the fault injected nodes. Even so the application kept running so we injected three more infinite pings on nodes 36, 37 and 41 marked as red. This last fault injection caused the application to fail.

Looking at the event log in the figure 10 we can actually see, compare and validate our predictions. Also we can use a tool from Cbench to validate that the experiment really failed. See figure 11.

```
Fri Aug 7 10:22:54 2009 network_fault started  workload = hpcc 32 nodes
Fri Aug 7 10:34:47 2009 injected fault=infiniting.pl to application=network_fault on node=node10
Fri Aug 7 10:34:59 2009 injected fault=infiniting.pl to application=network_fault on node=node20
Fri Aug 7 10:35:16 2009 injected fault=infiniting.pl to application=network_fault on node=node32
Fri Aug 7 10:35:39 2009 injected fault=infiniting.pl to application=network_fault on node=node40
Fri Aug 7 11:00:38 2009 injected fault=infiniting.pl to application=network_fault on node=node36
Fri Aug 7 11:03:27 2009 injected fault=infiniting.pl to application=network_fault on node=node37
Fri Aug 7 11:04:52 2009 injected fault=infiniting.pl to application=networkFault on node=node41
Fri Aug 7 11:08:51 2009 network fault failed  workload = hpcc 32 nodes
```

Fig. 10. Event log of experiment network_fault

**Parse Summary:**

----------

Total Files Parsed = 2
Total Jobs Parsed = 1

**Job Status Summary:**

----------

ERROR(STARTED) = 1
Overall Job Success = 0.00%

Fig. 11. output_parse Cbench script validates failure
8 Conclusion
During this research we can conclude a great success on the educational phase. Learning about fault tolerance, deeper knowledge in high performance computing, monitoring software and Perl scripting where just a few things that can be mentioned. Besides the educational phase, the system efficiently automates the steps of: fault injection, application start, event log, data gathering/monitoring start and shutdown.

9 Future work
Our future work would concentrate on exploring more fault injection techniques, exploring the integrity of the fault injections made and extended research on more complex HPC systems. Also this project is planned to be continued as a Senior Design Engineering, in which stage would be to build a complete package and integrate it to OSCAR. OSCAR (Open Source Cluster Application Resources) OSCAR is a snapshot of the best known methods for building, programming, and using clusters. It consists of a fully integrated and easy to install software bundle designed for high performance cluster computing [13].

Acknowledgement
I would like to thank my mentors for the opportunity to work on this project. Also, I would like to thank Thomas Naughton for his unconditional and continued support. The persons who wrote my recommendation letters: Prof. Idalides Vergara and Dr. Yahya Masalmah; without their support this opportunity would not be possible. Finally, special thanks go to Debbie McCoy, who made provisions for this research experience.

References
2009, Tennessee.


