HPC, Grids, Clouds:
A Distributed System from Top to Bottom

Brennon York **Naveed Alam**

CSCI – B534 Distributed Systems
School Of Informatics and Computing
Indiana University Bloomington

Introduction:

With the growth of the Internet and the abundant data deluge, the need for large systems to process such data becomes extremely important. Using supercomputers may help alleviate the problems but it too has its limits with prohibitive costs. Processing such large amounts of data requires a network of systems. Depending on the operations, the systems may be dispersed at different geographical locations and may also be diverse in their architecture and setup. Such distributed systems are increasing in popularity due to low costs, easy access, and faster processing. As Tanenbaum defines, "a distributed system is a collection of independent computers that appears to its users as a single coherent system."

This project aims to understand not only the working of some distributed systems, but also the technicalities of running multiple jobs on such machines. Search engines are a very good example of a distributed system and the tasks that such a system can efficiently perform. Therefore, understanding a distributed system begins from identifying and learning some of the tasks that can be performed on them.

Over the semester we have worked on multiple sub-projects dealing with different aspects of distributed systems. The goals of the projects are:

- Implement the popular page rank algorithm by Larry Page and Sergey Brin [2] as a linear program.
- Implement the page rank algorithm using MPI (Message Passing Interface) to run on a cluster.
- Determine speedup in the performance of the page rank implementation on bare-metal nodes and virtual machine instances while varying the number of processes and the size of input to the page rank program.
- Implement a resource monitoring tool using the Narada Brokering tool to monitor CPU and Memory usage on nodes.
- Build scripts to automate dynamic provision between bare-metal and virtual machine environments, run the message brokering software and start the page rank program.

Architecture and Implementation:

This section discusses the implementation details of each goal.

Page Rank Algorithm: The goal of the page rank algorithm is to determine the probability of selecting a particular page. The page rank algorithm works by first determining the initial probability of each web page in the input dataset. A web page that has more links leading to it from other pages has a higher probability of getting clicked thus, for each page, its intermediate page rank is calculated as the sum of the probabilities of every page that has output link to web page in question. The exact formula is as follows:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

Next, the probability of a user selecting a random page or not continuing further browsing is taken into account using a damping factor *d* and the page rank is calculated. This is averaged over multiple iterations to get a more accurate page rank probability. For MPI page rank each process computes page rank for a certain number of pages from the input dataset.

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

Linear Page Rank Implementation:

Program Flow: The Program begins in main accepting input from user.

1. Current system time is noted. An object of class *SeqPageRank* is created.
2. The *StartHere()* method of *SeqPageRank* is called passing to it the arguments obtained from the user.
3. In the *StartHere()* method, the arguments are checked and if incorrect an error message is displayed and program is exited.
4. If all arguments passed are correct then call *ReadInputFile()* method.
5. The *ReadInputFile()* method takes as argument the filename and reads every page and its outgoing links into the *pageLinks* hash map.
6. Next, in the *StartHere()* function the initial page rank probability is calculated and stored in the *pageRank* hash map.
7. For n iterations the page rank for all pages is calculated. The value of n is determined from input arguments to the program.
8. During each iteration, page rank is calculated in the *CalculatePageRank()* method.
9. The *CalculatePageRank()* method has an *intermediatePageRank* hash map and a variable for recording the dangling value.
10. For each page, its page rank value is calculated by averaging the probability of reaching that page through its incoming links. This value is stored in the *intermediatePageRank* hash map.
11. If a page does not have any outgoing links then its page rank value from previous iteration is stored in the dangling value variable.
12. The dangling value is averaged over the number of pages and the resulting value is added to every value in the *intermediatePageRank* hash map.
13. Next, the final *pageRank* value is calculated using the values in the *intermediatePageRank* hash map using a damping factor. The value os damping factor is obtained as argument to the program. The *pageRank* hash map is updated with the new page ranks from the current iteration.
14. After calculating page rank for n iterations, the results are output to a file by calling the *WriteToFIle()* method.
15. Next, the *Top10()* method is called which sorts the page ranks in descending order and outputs that top ten values to the Top10.txt file.
16. The control returns to the main function where the system time is noted and the time taken for the program is printed to output before the program is exited.

MPI Page Rank Implementation:

The aim is to divide the input dataset into groups of vertices where each group is assigned to one

process. Four files are used for calculating page rank; mpi_main.c, mpi_io.c, mpi_pagerank.c and pagerank.h. The files mpi_io.c and pagerank.h have been handed over with the project folder with the remaining files being authored by the members of B534 Group 11.

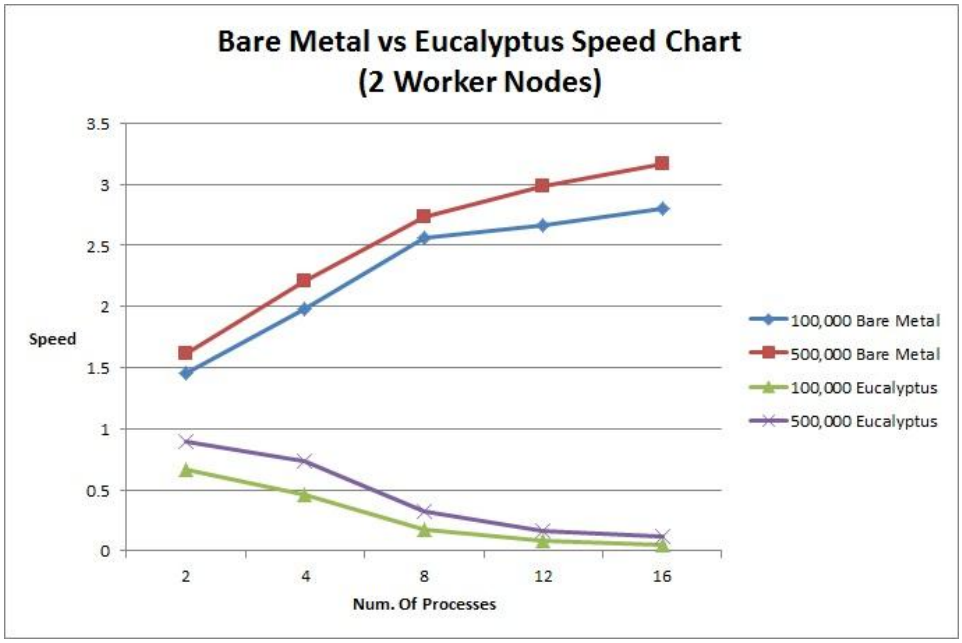
mpi_main.c: Initialize MPI and the number of processes stated on the command line. All arguments are parsed by all the processes. Each process then calls the mpi_read function in the mpi_io.c file. After the mpi_read function, each process has an adjacency matrix that contains the urls that are to be ranked and an am_index matrix that maps URLs to their location in the adjacency matrix as well as the number of outgoing links each URL has. Next, each process computes the initial rank values and stores them in the rank_values_table. Since, each process is allocated a chunk of the adjacency matrix and am_index, the values of the am_index need to be adjusted to map to the smaller allocated adjacency matrix. Then the mpi_pagerank function is called where each process computes the page rank values. After, page rank computation has completed by all processes, the process with rank 0 writes all the values to a file by calling the mpi_write function in mpi_io.c

mpi_pagerank.c: Each process first allocates memory for the local_rank_values_table and old_table which contain the current and previous rank values respectively. The rank probability of each page is calculated and added to the outgoing links of that page. Next, the rank value is approximated using the damping factor of 0.85. Only process with rank 0 computes a delta value using the difference of the newly calculated page rank values and rank values in the old_table. At the end of each iteration the delta value is calculated by process 0 and broadcasted to all other processes. If the delta value is less than the threshold value or the number of iterations has completed the page rank computation ceases.

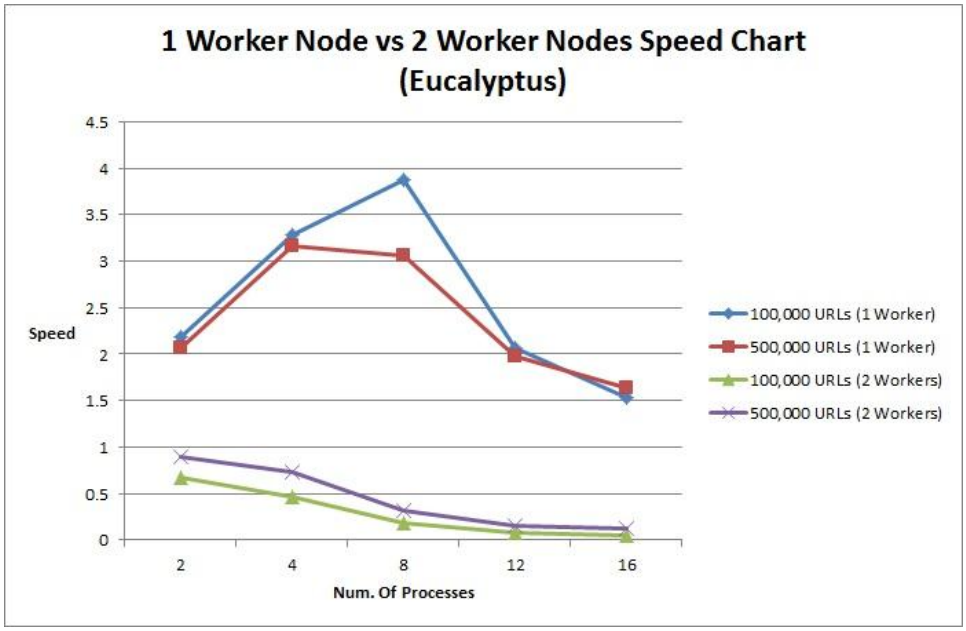
Speedup Performance:

The aim of the project is to determine the performance comparison between bare-metal and eucalyptus virtual instances at the Future Grid system at Indiana University Bloomington. The networking architecture of bare-metal and eucalyptus is very different and is apparent in the results. Bare-metal nodes are connected by high speed fiber-optic links while eucalyptus nodes are connected through 10 gigabit links.

To begin we will first describe each parameter used within our performance experiment. Our instance class was a c1.xlarge for the Eucalyptus virtual machine. We used two of these to keep our data sets equivalent with two worker nodes and a total of sixteen cores. There is also a comparison of worker nodes on Eucalyptus later as we found anomalies in our data. A static value of two was used for the number of worker nodes within every experiment. The size of our dataset ranged from 1,000 to 500,000 URLs with groups ranging from set 0 to set 4. This was done to show the performance degradation from the bare metal nodes as well as the Eucalyptus virtual machine to gain a better understanding of their scalar rates as more data is fed in. We decided to use a dynamic number of processes for this assignment to allow for quantifiable performance, not only at the OS level, but also on a per node level. The number of processes we used ranged from 2 up to 16 with jumps of two in between. For the threshold value we kept that static throughout all experiments at 0.000001. Some of our experiments did halt because of the threshold value. Additionally, the iterations stayed constant as well at 10. These last two variables remained constant to ensure equal performance analysis over the entire range of tests.



This graph demonstrates the speed comparison from bare metal to Eucalyptus.

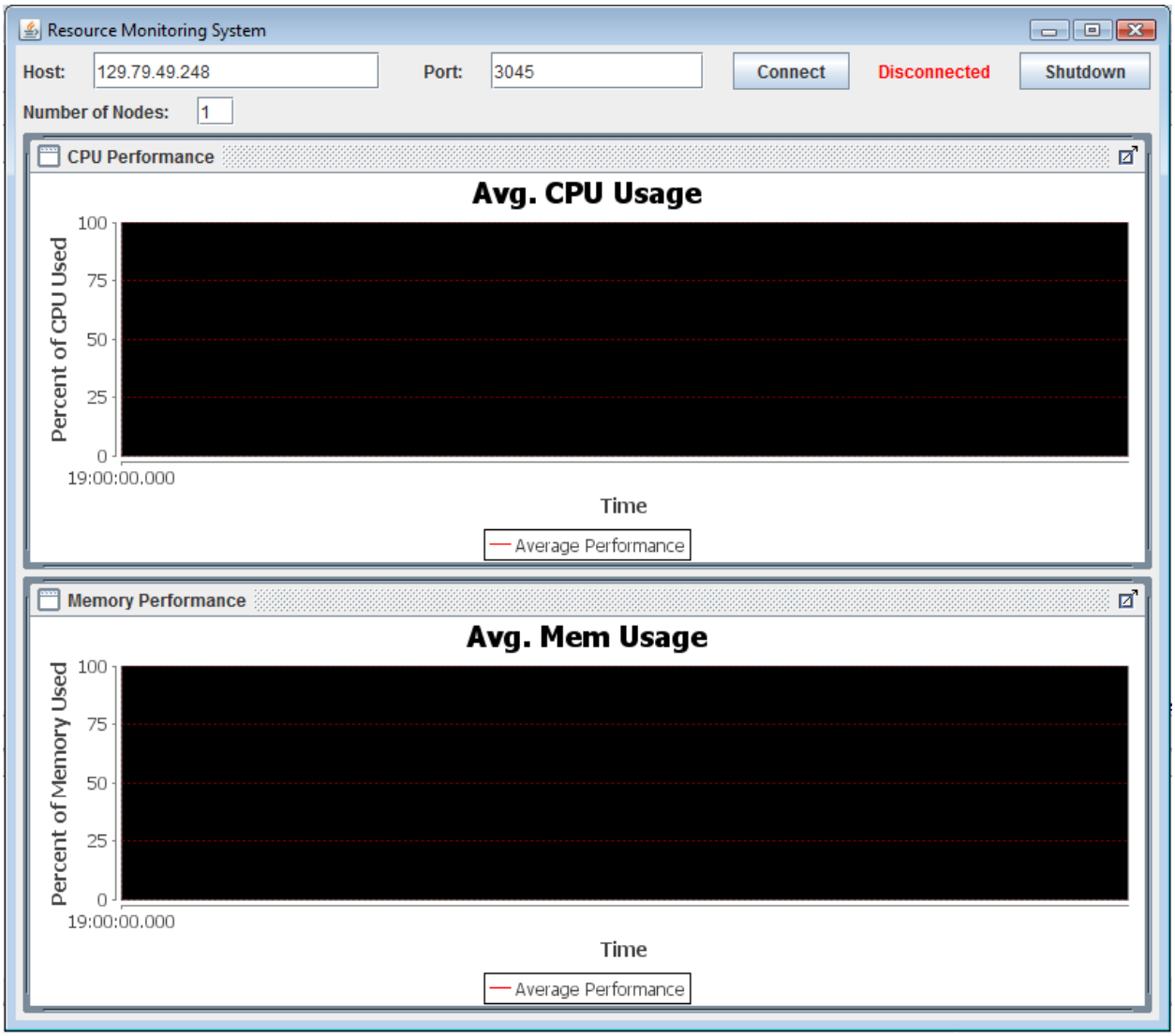


This graph shows the difference between 1 and 2 worker nodes.

Resource Monitoring System:

For this assignment we were supposed to implement a monitoring daemon and graphical user interface (GUI) for a brokering system, specifically Narada. This distributed system allows for publish / subscribe methodology in which monitoring daemons (programs) will publish specified data to the broker while the GUI subscribes to the broker to listen for these messages. The broker then acts as the unifying layer allowing for total autonomy between the two separate processes. This essentially acts in the same manner as the Internet layer (Layer 3) in the OSI architecture. Multiple different daemons and monitoring systems can be launched with no need to worry about how to deal with the interface displaying the data. Conversely it allows for multiple interfaces to subscribe to any number of specific messages from Narada to gather only the relevant data without the worry of daemon design and

message format.



Dynamic Provisioning System:

The dynamic provisioning system of future grid is utilized to dynamically provision/switch between bare-metal and virtual node instances. Dynamic provisioning allows on-demand resource allocation and instantiation is done on-the-fly. The Future Grid dynamic provisioning system uses Xcat, Moab and Torque job scheduler. The dynamic provisioning system architecture as defined in the project specification is shown below.

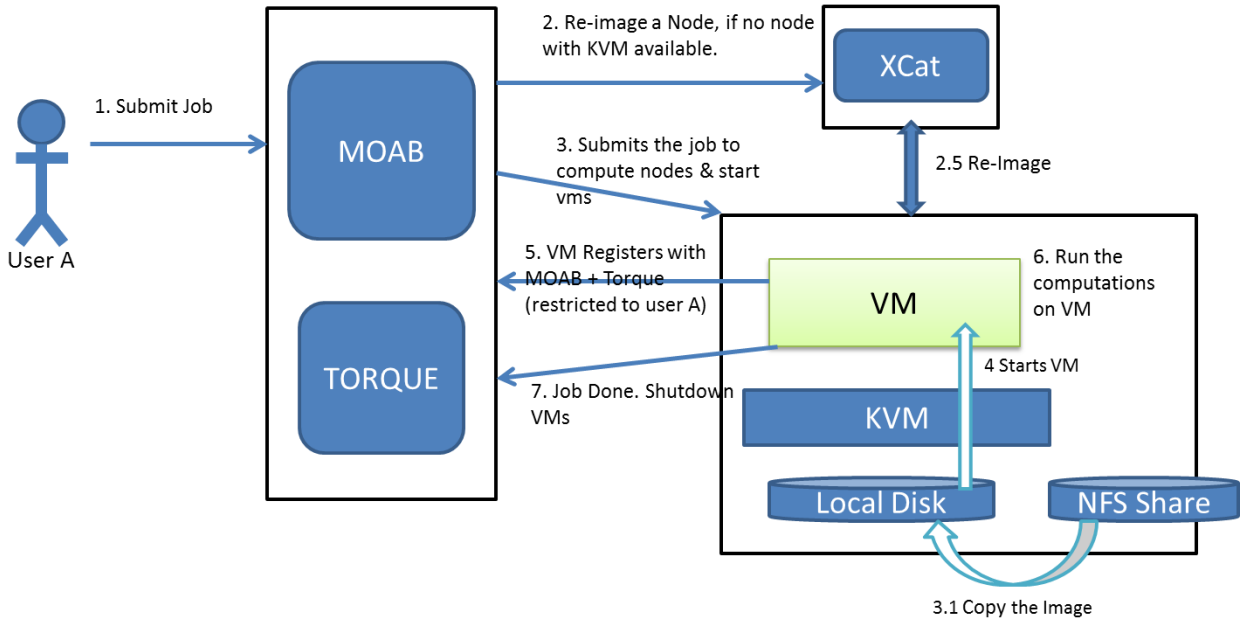


Figure: Architecture of Dynamic Provisioning System

We need to implement batch scripts to the system which will be placed in a queue. The script is responsible for allocating the resources. Once the compute resources are obtained, commands are executed to run tasks on obtained nodes.

Experimentation and Results:

Linear Page Rank

Implementation: Implemented in Java. The program is not multi-threaded and therefore does not need any special systems to run on. The command to run the program is
java [input file path] [output file path] [iterations] [damping factor]

MPI Page Rank:

The MPI program needs to be run on a cluster.
Implementation: It is implemented in C using MPI library.
First, check the MPI version on the cluster
mpi-selector -list
Next, choose the MPI version using the command
mpi-selector –set [mpi-version]

Program Compile: make
Program Execution:
mpirun -np [no. of processes] mpi_main -f [input file] -n [no. of iterations] -t [threshold value]
Eg: mpirun -np 2 mpi_main -f pagerank.input -n 5 -t 0.001

loop=0 delta=0.029254
loop=1 delta=0.010129
loop=2 delta=0.004182
loop=3 delta=0.001466
loop=4 delta=0.000612
loop=5 delta=0.000182
loop=6 delta=0.000045

loop=7 delta=0.000015
loop=8 delta=0.000007
loop=9 delta=0.000003

Speed Up Performance

Implementation and Setup: To run on bare-metal, we first obtained a set of worker nodes on futuregrid using the command:
$ qsub -I
Load the environment using the command:
$ module load intel
$ module load intelmpi
Allocate nodes using the command:
$ qsub -I nodes=[numOfNode]:ppn=[CpuPerNode],walltime=[hh]:[mm]:[ss]
Next, for running the program
$ mpirun -hostfile $PBS_NODEFILE -np [No. of processes] mpi_main -i [pagerank.input file] -n [iterations] -t [threshold]

To run on eucalyptus, first allocate virtual machines on futuregrid by the following steps in the folder provided during registration:
$ module load euca2ools
$ source eucarc
If key pair does not exist, create one
$ euca-add-keypair johnny > johnny.private
$ chmod 600 johnny.private
Create security group is not already done so,
$ euca-authorize -P tcp -p 22 –s 0.0.0.0/0 default
To allocate an instance,
command: euca-run-instances -k [public key] -t [instance class] [image emi #]
Finally, login into each instance and copy the key file to ~/.ssh_rsa and the program to the root directory. On one of the nodes create the file 'nodes' containing the list of internal IP addresses of each instance and run as
mpirun -hostfile nodes -np [No. of processes] mpi_main -i [pagerank.input file] -n [iterations] -t [threshold]

Results:
As you can see from the bare metal figures, the performance gains tend to wane at around eight processes. This means that running any more than eight processes for this type of data will most likely net you marginal performance increase while taking up more compute time than actually needed. This is something that should be especially worried about on an academic distributed compute system such as FutureGrid. When comparing the speed up with that of the Eucalyptus virtual machine one can see an immediate gap in performance. We noticed that speed actually decreased when running on the virtual machine. There are multiple explanations for this. First, there is the overhead cost of the virtual machine layer that all data must travel through that was not present when running on the bare metal. Second, the Eucalyptus environment on FutureGrid seems to be prone to user interruptions and scheduling issues, both of which could affect performance levels. Third, as explained in class, the Eucalyptus virtual machines can only support up to a 10Gbps link while the bare metal can handle the entire bandwidth of the fiber connection (40Gbps) thus creating a potential bottleneck when handling such large data sets. Last, as we will go into in more detail on the next graph, there seems to be an issue when working with two nodes as well. We believe this to create a message passing overhead

between the two nodes and add an additional bottleneck in the expected performance gains.

In this second graph we compare the speedup efficiency between using a single worker node versus two worker nodes. The key here is the drop off at eight processes for the single node. Since we used a c1.xlarge class the node came with eight cores. We attribute the single node speed up to the fact that all data processing stays within the same node without any message passing. Once the 12 and 16 process calculations enter, even though all data stays within the single node, processor scheduling efficiency now comes into play. This scheduling issue denotes the quick drop in performance after eight processes. Even though all of these issues seem to affect Eucalyptus, the best performance gain comes from it at eight processes with a single worker node, besting the bare metal data for 100,000 and 500,000 URLs.

To conclude, FutureGrid Eucalyptus and bare metal nodes were both great opportunities for us to learn and understand the complexities and issues regarding distributed systems. The bare metal nodes seemed to be much easier to get used to as they were much more accessible and did not require the scheduling that FutureGrid Eucalyptus did. On the other side FutureGrid Eucalyptus, although complex, was an amazing demonstration of how virtual machines work within large data centers. Our final results showed that Eucalyptus, if used in a single worker node scenario, seems to be the quickest up to eight processes. After that bare metal works better, although performance gains are minimal as processes continue to increase. The comparison between one and two worker nodes shows that the primary overhead for data-intensive calculations, under Eucalyptus, is the message passing between virtual machines.

Finally, it seems that the largest bounding factor to performance is the number of cores on a given node. If the calculations can stay within the node then performance seems to increase at a near-linear rate. Once the data crosses the barrier of the nodes it becomes bounded by the fiber tying them and can decrease overall gains.

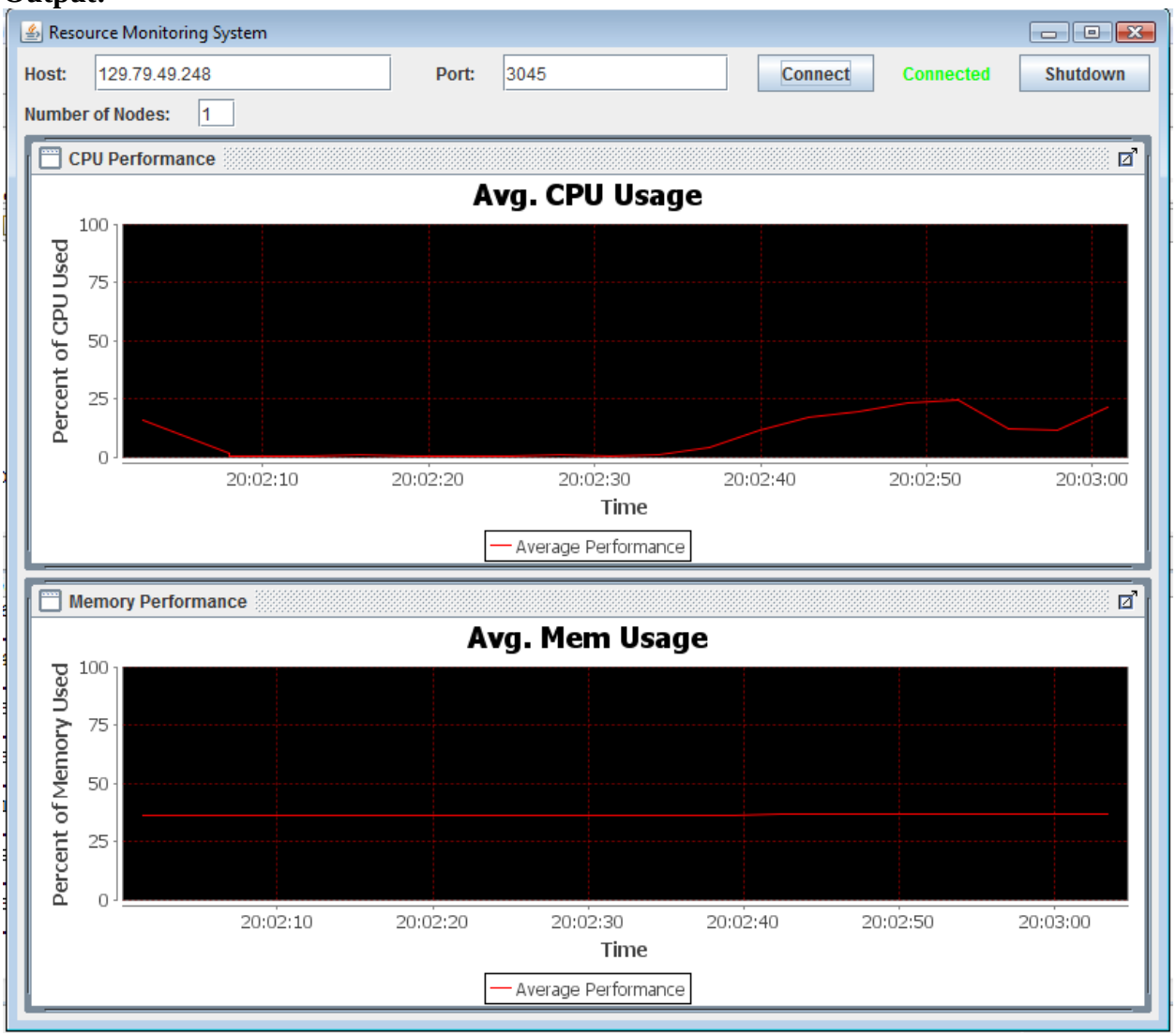
Resource Monitoring System:

Implementation: To accomplish this goal we simply utilized the most basic of monitoring and display functions. Some were chosen because of time constraints while others were chosen because of the inherent nature of the problem description. The latter is detailed below. Currently we run one monitoring daemon which gathers CPU load (in percent) and memory load (also in percent) through the deployment of two threads. These threads then publish the data, as different topics to the Narada brokering system. The graphical interface allows one to enter the IP address and port number of a Narada broker to subscribe to. In addition, the number of nodes for that will be subscribing to the broker should be initialized so that the performance data can be averaged over the received values for all nodes. If any daemons are publishing to the broker then the content is summarized as a graph over time for both CPU performance and Memory performance. The programs are implemented in java. The JFreeChart library is used at the subscriber end for representing results as graphs. The Sigar library is used at the publisher to get CPU and memory information.

Compiling and running the subscriber
Compile - javac -cp [path to library files] Project3/src/*.java
Run - java -cp [path to library files] GUI.java

Compiling and running the publisher
Compile - javac -cp [path to jar files] IndividualPublisher.java
Run - java -cp [path to jar files] IndividualPublisher [IP of Broker] [Port of Broker] [No. of trials]

Output:



Dynamic Provisioning System:

Implementation: To handle the submission of the jobs from a simple script we needed to understand how to properly setup and automate the MPI execution. This began as a trivial task when needing to measure performance on a bare metal cluster. All that was needed were the original commands procured from the first assignment and then scripted. One key difference was that, because we utilize the Torque system, we were able to pass specific directives from the head node. Some features include the number of nodes needed, number of processors needed, and the estimated time to complete the batch job. Torque then graciously hands this information off to the designated node it finds to run the job. When dealing with the virtual machine implementation things became a little trickier. One of the biggest hurdles was in understanding how to simulate a remote procedure call (RPC). We use the term simulate because our implementation piggy-backs off of the default SSH command to remote into the node and execute a given command. Once the virtual machine has been started, through scripts given to us for the project, we can then perform our RPC calls and secure copies for data transfer. Another, smaller, hiccup came when handling local libraries. It was soon realized that, for the local virtual machine, to understand where the shared libraries were the program had to be recompiled. The entire sequence of events is as so:

1. Define the preprocessing directives to the Torque submission engine.
2. Run the /usr/local/bin/start_vms command to procure a virtual machine.
3. Run the /usr/local/bin/wait_for_vms command to wait for the virtual machine to become active for use.
4. Copy over all code necessary (Publishing daemon and MPI Page Rank).
5. Recompile the code locally.
6. Run the Publishing Daemon.
7. Run the MPI Page Rank Program.
8. Shutdown the virtual machine and clean up any excess files.

An Example output follows:
Starting VM's on i76
Staging the VM on i76
starting VM on i76
Domain rhels5.5 created from /tmp/rhels5.5.xml

IP addresses of the VM's
149.165.146.209
You can also get the ip addresses of VM's from the VM_NODEFILE.
Waiting for the VM's to be reachable
149.165.146.209
Shutting down VM's
Domain rhels5.5 destroyed

VM Job finished

Once all of these steps were completed one should have a working monitoring publisher daemon running with a complete pagerank.output file.

Conclusion:

Starting from the linear page rank implementation, we have learned and implemented a parallel page rank program to run on a cluster. Having run and tested the MPI program on a cluster, we moved to determining the performance of our implementation on bare-metal and eucalyptus virtual instances of the future grid system. The results obtained helped gain a better insight of two clusters. Performance improved on bare-metal nodes with increasing number of processes keeping with the number of cores allocated. However, on eucalyptus, there was a speed-down rather than a speedup. Interestingly, a speedup was observed on eucalyptus when running a single extra-large node up to 8 processes. Thus, we have concluded our results attributing the disparity in results between bare-metal and virtual instances to the network connection, allocation of users to virtual nodes and the scheduling of processes on each cluster. Next, we implemented a resource monitoring system that can monitor the resource usage on multiple nodes by subscribing and publishing to the Narada Brokering System. The implementation shows real-time resource utilization of resources being monitored. Finally, the batch scripts dynamically allocate compute resources and schedule the processes to run on these resources. Overall, we have understood and learnt the structure and aspects of different parts of distributed systems.

Acknowledgments

We are grateful to Professor Judy Qiu for having given this opportunity to explore distributed systems

and thus having empowered us with knowledge to better understand building and provisioning for such systems. We are also thankful to the associate instructors Ikhyun Park and Pairoj Rattadilok for guiding us through the projects.

References

1. Andrew S. Tanenbaum and Martin Van Steen, “Distributed Systems Principles and Paradigms”, second edition.
2. Sergey Brin and Lawrence Page, “The Anatomy of a Large Scale Hypertextual Web Search Engine”, Proceedings of the Seventh International World Wide Web Conference, April 1998.
3. http://en.wikipedia.org/wiki/Markov_chain
4. http://en.wikipedia.org/wiki/Adjacency_matrix
5. http://en.wikipedia.org/wiki/PageRank
6. Ganglia, http://ganglia.sourceforge.net/
7. Nagios, http://www.nagios.org/
8. Amazon Cloudwatch http://aws.amazon.com/cloudwatch/
9. NaradaBrokering, http://www.naradabrokering.org/
10. ActiveMQ, http://activemq.apache.org/
11. Sigar Resource monitoring API,
12. http://www.hyperic.com/products/sigar http://sourceforge.net/projects/sigar/
13. JFreeChart, http://www.jfree.org/jfreechart/
14. TORQUE Resource Manager http://www.clusterresources.com/products/torque-resource-manager.php
15. KVM Hypervisor http://www.linux-kvm.org/page/Main_Page
16. libvirt: The virtualization API http://www.libvirt.org/
17. Torque Qsub: http://www.clusterresources.com/torquedocs21/commands/qsub.shtml#I
18. Torque Job submission: http://www.clusterresources.com/torquedocs/2.1jobsubmission.shtml
19. B534 lecture slides from Oncourse.