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

Purshottam Vishwakarma, Bitan Saha
M.S. Computer Science
Indiana University
{pvishwak, bsaha}@indiana.edu

***Abstract* — High performance distributed systems are amongst the most complicated Computer systems to design and implement. The complexity largely stems from the various components of such a system which need to interact seamlessly with one another and still be able to function through intermittent failures in system components. In this project we have demonstrate distributed system capabilities in the context of page rank algorithm. We compare Sequential page rank algorithm with the parallel version of the algorithm and present performance results. We also implemented the resource monitoring and dynamic provisioning of the distributed system to observe the real time performance of our page rank program. The entire project has been executed in 4 phases.**

***Keywords- Page Rank*, Dynamic Provisioning, Resource Monitoring.**

I. INTRODUCTION

PageRank algorithm for internet search took the world by storm for solving the large data set problem i.e. efficiently searching the web. But with increasing number of web pages it was even more important to optimize the algorithm by parallelizing it. Needless to say that search has become the most important part of our lives. This provides us enough motivation in academia to implement both sequential and parallel version of page rank algorithms and practically learn the distributed aspects of large scale computing and how it can be optimized for efficiency. It is not only important to implement the algorithm but also equally important to evaluate its performance i.e. CPU and memory consumption real time. In the following sections we address the problem of how efficiently we can increase the efficiency of the algorithm i.e. by what factor we increased the performance of pagerank algorithm by parallelizing it using MPI. In the third phase, we added a monitoring daemon on the academic cluster which evaluated CPU and memory usage of our PageRank program and published this information to an ActiveMQ Message Broker. We also set up a ActiveMQ client which subscribed to the published information and used it to draw a performance chart of CPU and memory usage over time. In the last phase of the project, we developed batch scripts to fully understand how a shared HPC cluster is used by a number of requesting processes. The scripts were submitted to a queue which when ready would be executed to start our daemons on all the nodes, run our PageRank program and publish CPU and memory usage for the period of our job. A brief description of each of these phases is described in the remaining sections.

II. IMPLEMENTING PAGERANK USING MPI

The URLs and their associations are defined in a text file which is provided as input to our page rank program. We tested our results with both 10 URLs and 100 URLs. We create the adjacency matrix from the input file provided.

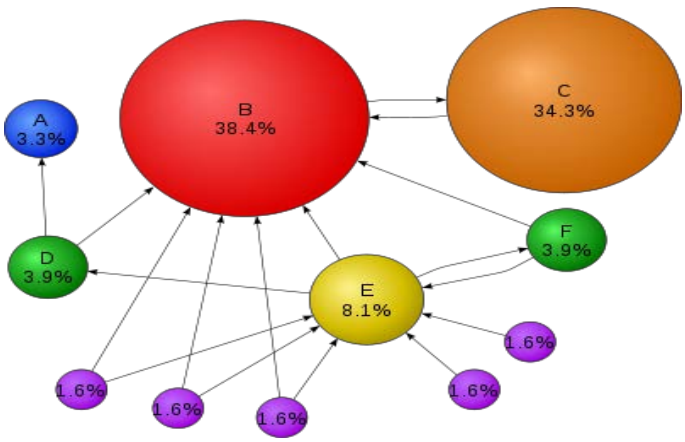


Figure 1: Mathematical PageRanks for a simple network, expressed as percentages. Courtesy: Wikipedia

The algorithm splits the entire adjacency matrix among the number of processes specified. For example, the below adjacency matrix is split among 4 processes as shown in Figure 1.

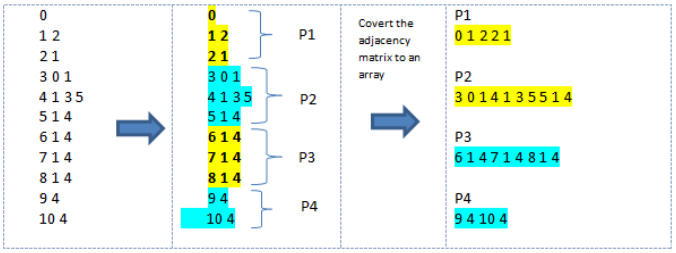


Figure 2 : Dividing the Adjacency matrix to different processes

- Each process keeps the track of the root node (1st column in the original adjacency matrix) using am_index data structure.
- Each process calculates the associated page rank of the outbound nodes and dangling values of the root node. After calculating them each process sends these two values (pagerank_values_table and the dangling values) to MPI and it sums up the values received from all the processes and then sends back the aggregated values to all the processes. Thus each process now has the aggregated values of the pagerank.
- Similarly MPI_Allreduce() receives individual process dangling values, sums them up and sends back the aggregate values back to each process.
- Each process then adds the damping factor into the calculated value to get the final value of page rank.
- The above bulleted steps are performed for the specified number of iterations.

The diagrammatic representation of the MPI Page rank algorithm for iteration 1 is shown in figure 2.

Process 1	Process 2	Process 3	Process 4
[0] = 0.000000	[0] = 0.045455	[0] = 0.000000	[0] = 0.000000
[1] = 0.090909	[1] = 0.121212	[1] = 0.136364	[1] = 0.000000
[2] = 0.090909	[2] = 0.000000	[2] = 0.000000	[2] = 0.000000
[3] = 0.000000	[3] = 0.030303	[3] = 0.000000	[3] = 0.000000
[4] = 0.000000	[4] = 0.045455	[4] = 0.136364	[4] = 0.181818
[5] = 0.000000	[5] = 0.030303	[5] = 0.000000	[5] = 0.000000
[6] = 0.000000	[6] = 0.000000	[6] = 0.000000	[6] = 0.000000
[7] = 0.000000	[7] = 0.000000	[7] = 0.000000	[7] = 0.000000
[8] = 0.000000	[8] = 0.000000	[8] = 0.000000	[8] = 0.000000
[9] = 0.000000	[9] = 0.000000	[9] = 0.000000	[9] = 0.000000
[10] = 0.000000	[10] = 0.000000	[10] = 0.000000	[10] = 0.000000
Dangling value = 0.090909	Dangling value = 0	Dangling value = 0	Dangling value = 0

MPI_Allreduce

[0] = 0.045455
[1] = 0.348485
[2] = 0.090909
[3] = 0.030303
[4] = 0.363636
[5] = 0.030303
[6] = 0.000000
[7] = 0.000000
[8] = 0.000000
[9] = 0.000000
[10] = 0.000000
Sum_Dangling = 0.090909

Figure 3: After 1st Iteration of MPI Page Rank

The algorithm is presented in figure 4.

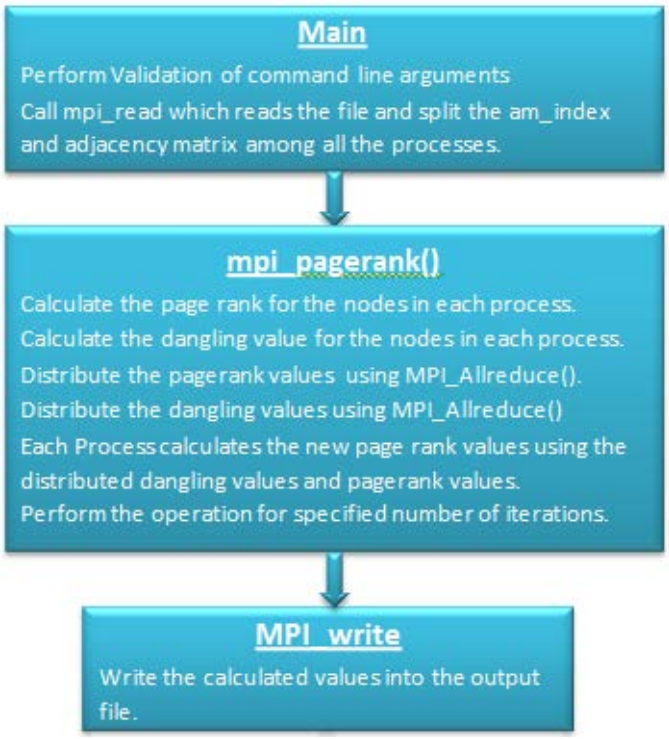


Figure 4: Algorithm for MPI Page rank.

Peformance analysis

We benchmarked our program with the sequential version of the program and found that the MPI version of page rank works faster than the sequential version. Table 1 shows the timings for the sequential version and the MPI version with 1000 nodes, 10 processes and 10 iterations.

Sequential Version of PageRank	1 s
MPI Version of PageRank (5 processes)	1 ms
MPI Version of PageRank (10 processes)	3 ms
MPI Version of PageRank (15 processes)	4 ms
MPI Version of PageRank (20 processes)	8 ms

Table 1: Performance Results of MPI Page Rank Algorithm

III. PAGERANK PERFORMANCE ANALYSIS ON FUTUREGRID AND EUCALYPTUS

One of the objectives of running page rank algorithm on cloud environment is to obtain gain in the performance of the algorithm based on several parameters like number of cores, number of nodes, number of MPI processes, number of iterations, number of URLs, etc. We performed several experiments with varying parameter values to best reflect the gain in performance. The experiments are performed on FutureGrid (Bare Metal) and Eucalyptus.

Parameters and Data Set

The experiments were performed with the following parameters remaining constant.

Damping factor = 0.85
Number of Iterations = 10

Below Parameters were changed with different values during the experiment.

Number of Urls: 1K, 10K, 20K, 30K, 40K, 50K , 60K, 70K, 80K, 90K, 100K, 500K, 1M and 2M

Number of Nodes: 1 Node (8 Cores), 2 Nodes (16 Cores)

Number of MPI Processes: 2, 4, 6, 8, 10, 12, 14, 16

Number of VMs: m1.large(2 CPU cores) : 2, 4, 6, 8
c1.xlarge(8 CPU cores) : 1, 2

Performance Graphs on FutureGrid (Bare Metal)

The first set of experiments was performed with 2 nodes (16 Cores) and all different number of URLs. Figure 5 shows the page rank computation time in milliseconds (ms).

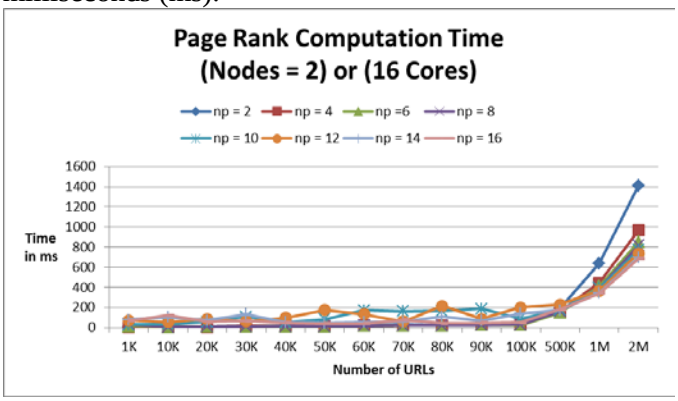


Figure 5. Performance Chart with 16 cores

The second set of experiments was performed with 1 node (8 Cores) and all different number of URLs. Figure 6 shows the page rank computation time in milliseconds (ms).

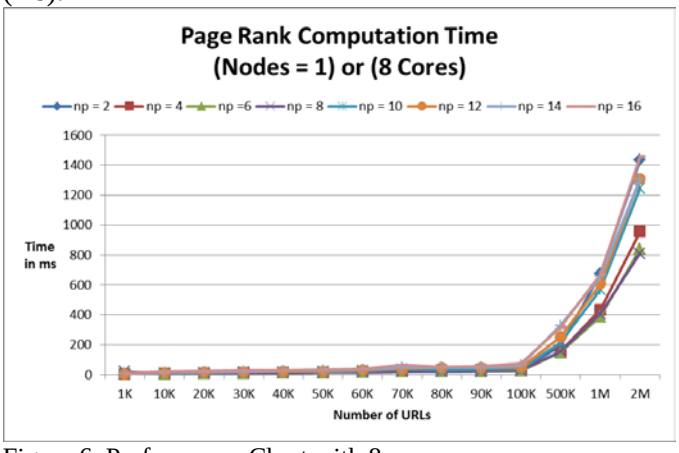


Figure 6. Performance Chart with 8 cores

When comparing the with respect to number of MPI processes, it is evident that the performance for np =8 is much better performance with np = 16 (highlighted with yellow in the table). This is due to the fact that there are too many processes for which there is a huge communication overhead. With np = 16, the master process has to collect data from 15 different processes. Hence, in comparison to np = 8, it spends more time communicating which leads to performance degradation. When comparing single node and 2 node scenario, two nodes performance is marginally better than single node scenario. But as the number of nodes increases, the performance might decrease because there are more inter-machine communication and network latency impacts the performance. In case of single node, all the cores are in the same physical machine and hence there is no network communication overhead

Speedup Graph with FutureGrid (Bare Metal)

We measured the performance gain when number of MPI processes is 8 and 16. The Speed up is given by the formula

$$\text{Speedup} = \frac{T_1}{T_p}$$

p is the total number of cores/processes

T_1is the execution time of the sequential algorithm

T_p is the execution time of the parallel algorithm with p cores/processors

Best Speed up we received is 3.11 times (1 node, 2M URLs and np = 8) and 3.67 times (2 nodes,2M URLs and np =16) when number of MPI processes = 6.

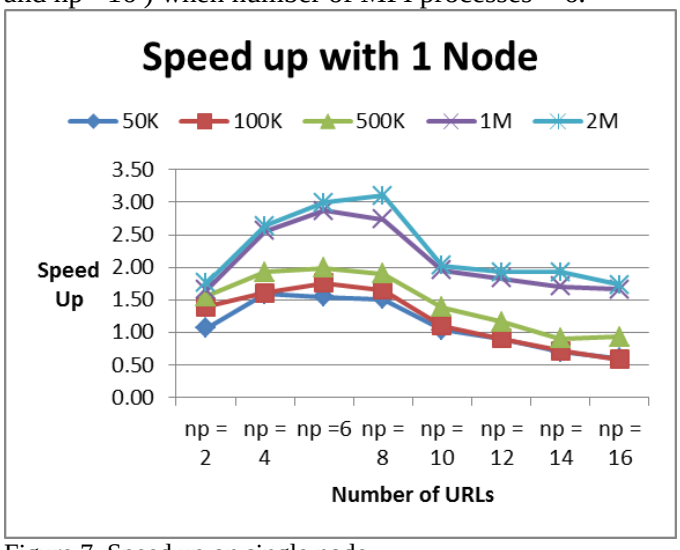


Figure 7. Speed up on single node.

In the Figure 7 and 8 the speed up is maximum when Number of URLs = 50K after which the speed up decreases. This is due to the fact that as the data size increases it takes more time to transfer the data across machines (2 nodes). In case of single node the speed up almost remains flat after 50K but marginally less than the speed up obtained with 2 nodes because the

computation task is spread across 2 nodes in parallel which makes it marginally faster.

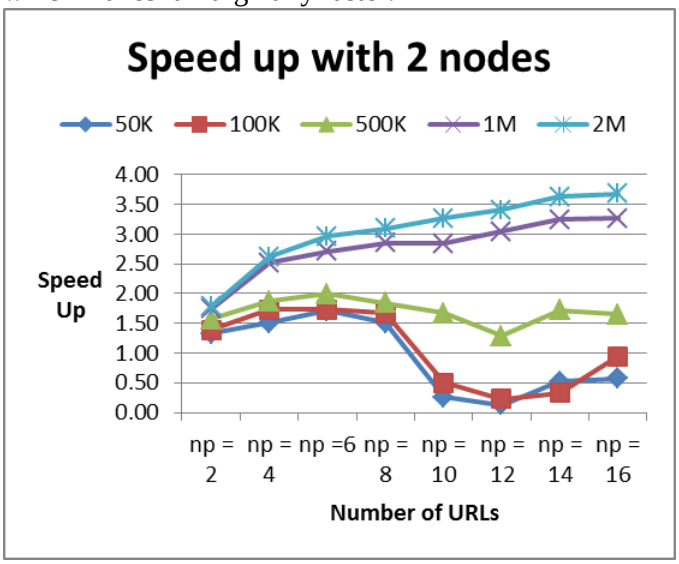


Figure 8. Speedup with 2 nodes

When speed up is compared with respect to number of nodes, for higher number of URLs, number of nodes = 2 gives better performance. This is because the computation is distributed among more number of nodes which leads to faster calculations of Page Ranks. When the number of URLs is low, it doesn't make sense to distribute the computations among larger number of processes because most of the time is lost in communication overhead which leads to low performance. We can see that the speed up graph for 1M and 2M URLs is almost increasing.

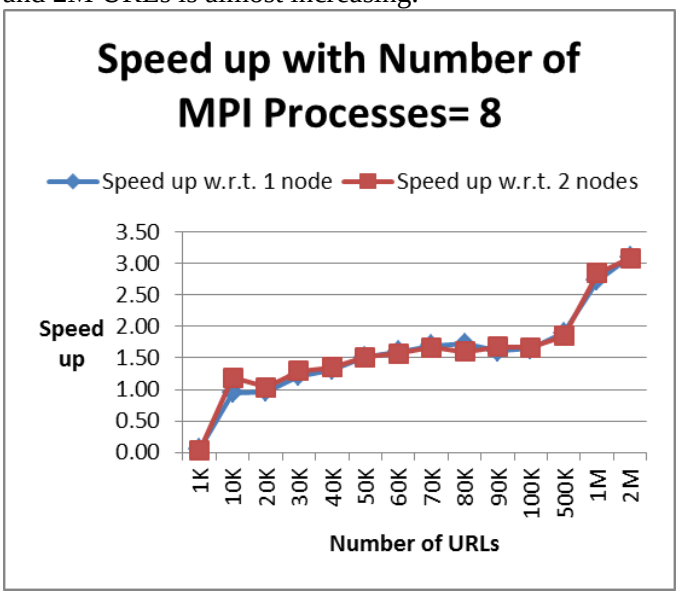


Figure 9 Speedup with 8 processes.

With 1M and 2M URLs and 1 Node: The performance decreases as number of processes go beyond 8. This is due to the fact that the single machine has only 8 cores. As the number of processes goes beyond the number of cores, the additional processes keep waiting for the CPU core to get executed. With np=8, each process is executing on each core, so no process waits for the CPU and the performance is the best.

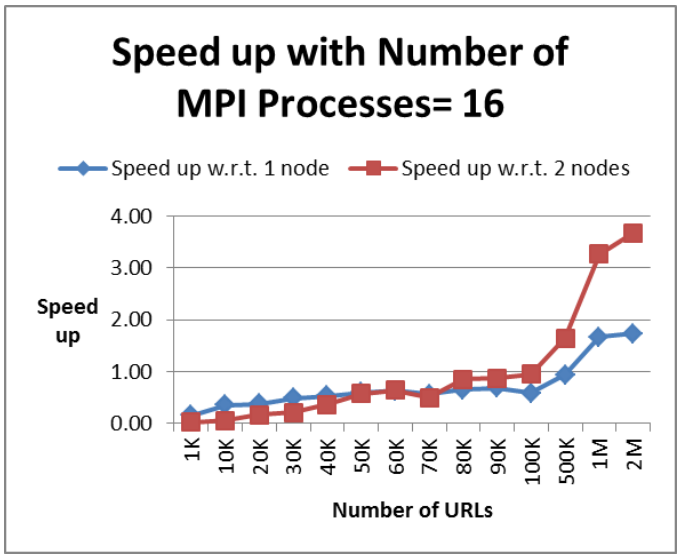


Figure 10. Speedup with 16 Processes

With 1M and 2M URLs and 2 Node: In this case the data set is huge and hence it becomes necessary to divide the work between different processes. Now we have 16 cores available. So if we create any number of MPI Processes less than or equal to 16, each process will get their own exclusive core to execute leading to higher performance. We should not forget the communication overhead involved between 2 nodes. When the data set is less, the communication overhead weighs more than the computation cost. Hence the performance is less for smaller data set i.e. Number of URLs < 50K. It is evident in the two graphs below:

Performance Graphs on Eucalyptus(Virtual Machine)

We ran MPI pagerank program with all the combinations of different number of VMs and different number of MPI Processes. The Graphs of all of them are shown below.

The Performance is best when the number of MPI Processes are less than or equal to the Number of Virtual Machines.

The reason is that when the number of MPI processes becomes greater than Number of Virtual Machines, the communication overhead between Virtual Machines is large leading to decrease in performance.

The MPI library assigns each process to each VM i.e. one to one, leading to better performance if the ratio of MPI Process and Virtual machines remains 1 or less.

When the Number of Processes per VM is more than 1, it increases the communication overhead and hence leads to decreased performance.

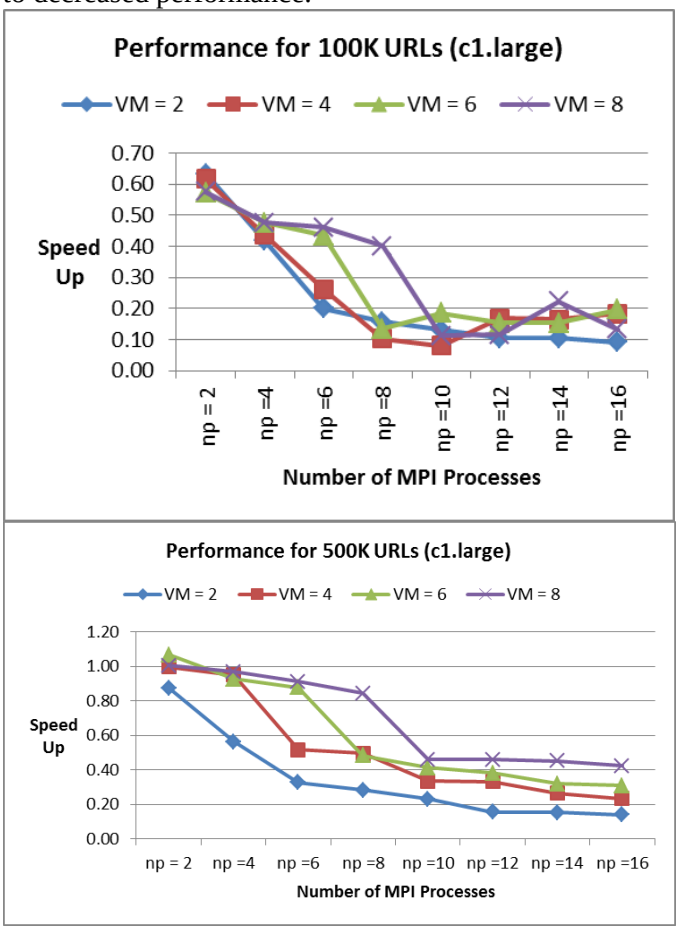


Figure 11 : Speedup with different URLs on Eucalyptus

When comparing the performances with 1 VM and 2 VMs, both on c1.xlarge, the speed up with 1 VM is higher than that of 2 VM. This is because the in case of 2 VMs there is a huge communication overhead between the VMs which is not the case with single VM. In single VM, all the processes remain in the same VM and hence there is no network communication overhead involved.

Also, the performance decreases with the increase in number of processes. This is because as the number of processes increase, the ratio of no. of cores to number of processes decreases which makes the other processes waiting for the CPU. Also with VM there are additional processes in the system waiting for the CPU. This also proves why the sequential program performance decreases in VM by 20% - 50% as compared to Bare Metal.

IV. RESOURCE MONITORING

For this part of the project, we implemented a system that monitors the CPU and memory utilization in a distributed set of nodes. The system supports the monitoring of bare metal as well as VM nodes running Linux OS's. Monitoring information is collected and aggregated through the message broker and summarized to display the overall CPU and memory utilization percentages using graphs.

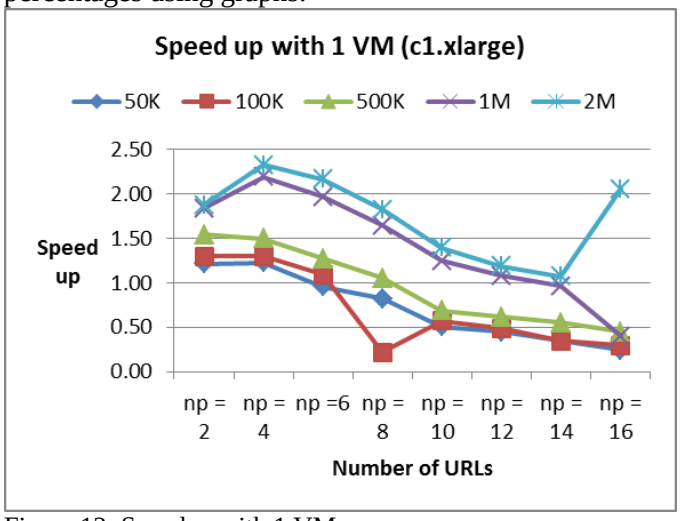


Figure 12: Speedup with 1 VM

We have chosen "Java" as the programming language to implement both Monitoring system and the Daemon. We are using "Active MQ" in non-persistent mode as our Message broker.

We used both topic as well as Queue for communication in our implementation.

Topic is used for control messages (Synchronization) whereas Queue is used for transferring data and registration messages.

Our Monitoring system is designed to support n (n>= 1) number of nodes.

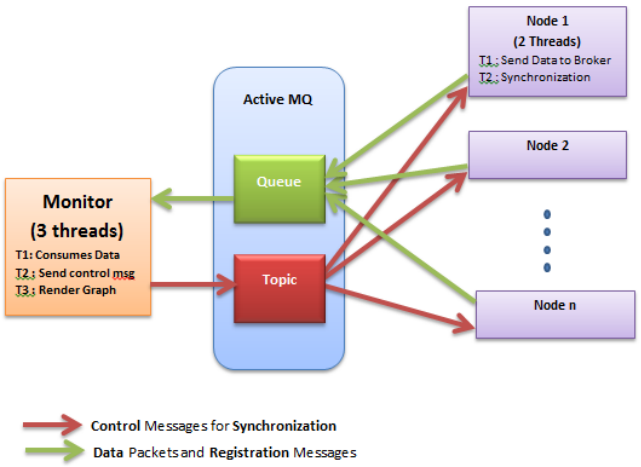
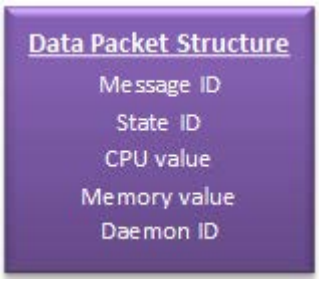


Figure 13. High Level Architecture of Resouce Monitoring

Each node runs a daemon to collect CPU and memory Usage and routes it to the monitor through the Queue for visual representation. Monitor uses Topic to send control

messages. New Nodes also use Queue to send registration messages to the Monitor. Monitor keeps a buffer for each node. So for n nodes it will have n buffers. One separate thread in the monitor reads all the buffers and renders the graph. After the data has been read it is cleared from the buffer.

Registration & Synchronization:
Below is the data packet structure. Each member in the structure is described below:



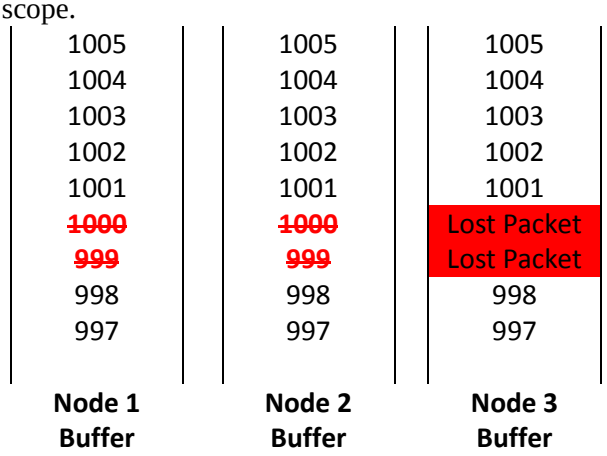
Message ID: This is an integer value used to identify the sequence number of the messages which is incremented with every data packet sent. The first data packet sent will have the message ID = 1 and subsequently incremented by 1.
State ID: This is used to identify the Phase of data transfer. The state changes every time a new node registers i.e. all the nodes will start sending data starting with message id 1 and this new state id.
CPU Value: This is the CPU usage at a given instance.
Memory Value: This is the Memory usage at a given instance.
Daemon ID: This is the combination of system name and IP address to identify each node uniquely.

Whenever a new node joins the monitoring system, it sends a registration message to the Monitor through the Queue. Once the monitor receives the registration message it clears the buffer of each Node and sends a synchronization message with a new State ID to all the Nodes including the newly added node to reset their Message ids to 1.

Hereon all the registered nodes will use this newly generated State ID along with message ID starting from 1 to send the data messages. This indicates a new phase of data transfer between Monitor and the Nodes. If the monitor receives a data packet with old state id, those data packets are discarded. This mechanism is used to synchronize all the nodes.

Auto Correction:

We have incorporated auto correction feature in order to deal with the lost data packets. As an example, suppose there are 3 nodes in the system. Monitor will maintain 3 separate buffers for each node. The message received from each of the nodes are stored in the buffer corresponding to each node as shown the below diagram. Now suppose the monitor doesn't receive Message ids 999 and 1000 from Node 3. Note that we are using Active MQ with non-persistent mode. Hence requesting the broker to send the lost data packets is out of scope.



In such a scenario, we will discard the message id 999 and 1000 from all other node buffers. This entails we display the data from message id 1001 after 998.

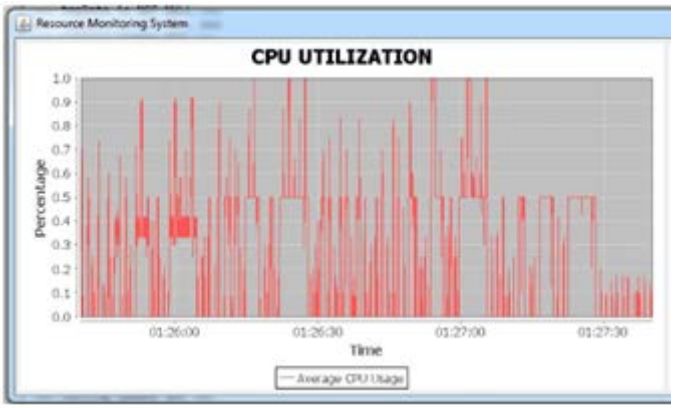


Figure 14a: CPU Utilization is maximum with number of MPI processes = 16 and 2 Million URLs.

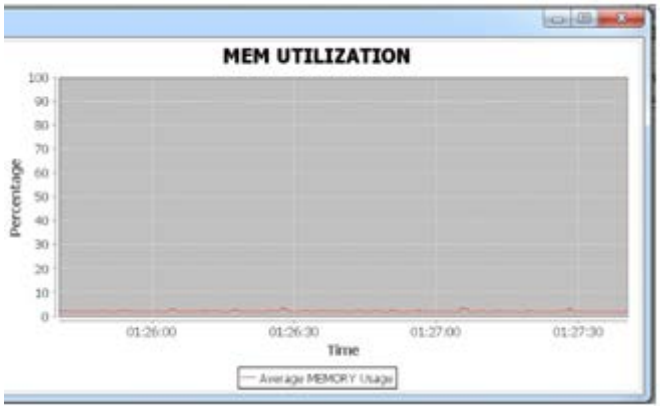


Figure 14b: Memory Utilization with number of MPI processes = 16 and 2 million URLs.

V. DYNAMICALLY PROVISION CLUSTERS ON ACADEMIC CLOUD

The last phase of our project was to be able to submit jobs to the academic cloud so as to simulate real world resource sharing. We had to set up dynamic scripts to start our daemons and MPI program and end gracefully (release all resources) should our job fail or finish. The FutureGrid team provided us the ability and to use on-demand resources in a shared academic Cloud environment – a user of this system could simply send a request with specifying their needs to the resource manager to obtain different kinds of computing resources, where the requested computing resources are deployed or instantiated on-the-fly. With the support of FutureGrid System Admin team, we built a Dynamic Provisioning system which can switch between Bare Metal and Virtual Machine compute cluster environments utilizing the XCat, Moab and Torque job scheduler

Dynamic Provisioning in combination with ActiveMQ was used in our project to monitor only PageRank Algorithm specifically.

Based on the information received from the monitoring infrastructure, users will programmatically switch/re-provision their nodes to another environment (eg: from Linux to Linux VM's). Following figure shows the interactions between each component within this system.

For our project, the jobs were subscribed under a special job queue (b534) in india.futuregrid.org.

A user can submit batch or interactive jobs. However, the focus of this phase of our project was using only Batch jobs. With batch jobs a PBS job script is submitted, written as a shell script, which will wait in the queue and get executed when the resources are available. Inside the job script, we could obtain a list of compute resources assigned to us, using $PBS_NODEFILE or $VM_NODEFILE (depending on if we used baremetal nodes or Virtual Machines).

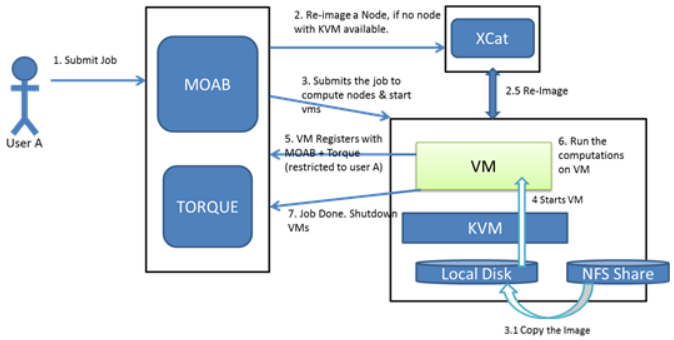


Figure 15.User interactions with dynamic Provisioning system

Scripts for BareMetal:

Following steps were followed to run the scripts on BareMetal.

1. Acquire the PBS nodes. Here we acquired i97 and i98 nodes which were assigned to us.
2. Output $PBS_NODEFILE contents (unique) to nodes file. In the nodes file we will have unique node ids. i.e. i97 and i98.
3. Loop each node in the nodes file.

a. If the node is the head node i.e. the first node in the file, start the daemon without ssh.

b. Else start the daemon using ssh (ssh $LINE "cd $HOME/B534/P2/DeamonCode;./startDeamon.sh &" &).

4. Load the mpi module using the command "module load openmpi"
5. Run the mpi program using mpirun.
6. Loop each node in the nodes file.

a. Close the daemons using ssh.

b. For head node close the daemon without ssh.

7. End bare metal job.

Appendix 1 shows the performance chart for Bare Metal.

Script for VM:

1. Acquire the PBS nodes. Here we acquired i97 and i98 nodes which were assigned to us.
2. Shutdown the VMs if there are any already running VMs and wait for 60 seconds to shut down.
3. Start the VMs using the script start_vms.
4. Output $VM_NODEFILE contents (IP addresses) to nodes_vm file. In the nodes_vm file we will have the IP Addresses of the Virtual Machines.
5. Loop each IP addresses in the nodes_vm file..
   a. Start the daemon using ssh (ssh $LINE "./startDeamon.sh &" &).
6. Start the mpi program using ssh on the Headnode using the below command.
   ssh $HEADNODE "module load openmpi;mpirun --mca btl_tcp_if_exclude lo,eth1 -hostfile nodes_vm -np 6 mpi_main -i pagerank.input.2M -n 500 -t 0.000001 >> mpi.out"
7. Loop each node in the nodes_vm file.

a. Close the daemons using ssh (ssh $LINE "killall -e java -u pvishwak &" &).

8. End job.

Appendix 2 shows the performance chart for Bare Metal.

VI. CONCLUSION

Over the course of this project, we have been exposed to parallel algorithms, hardware to support the parallelism and real-world distributed system frameworks. We've gained experience in running simple parallel jobs such as the MPI PageRank on academic

clusters to developing systems to monitor and collect statistics from such non-trivial systems.

The use of Futuregrid's academic cluster was an excellent introduction to using shared HPC resources. The use of the Dynamic Provisioning system enabled us to appreciate the value of sharing limited resources. We are very pleased with the work we have been able to do and the results we have been able to put up over the course of the last four months. We look forward to applying our knowledge of distributed systems in real-world/industrial environments.

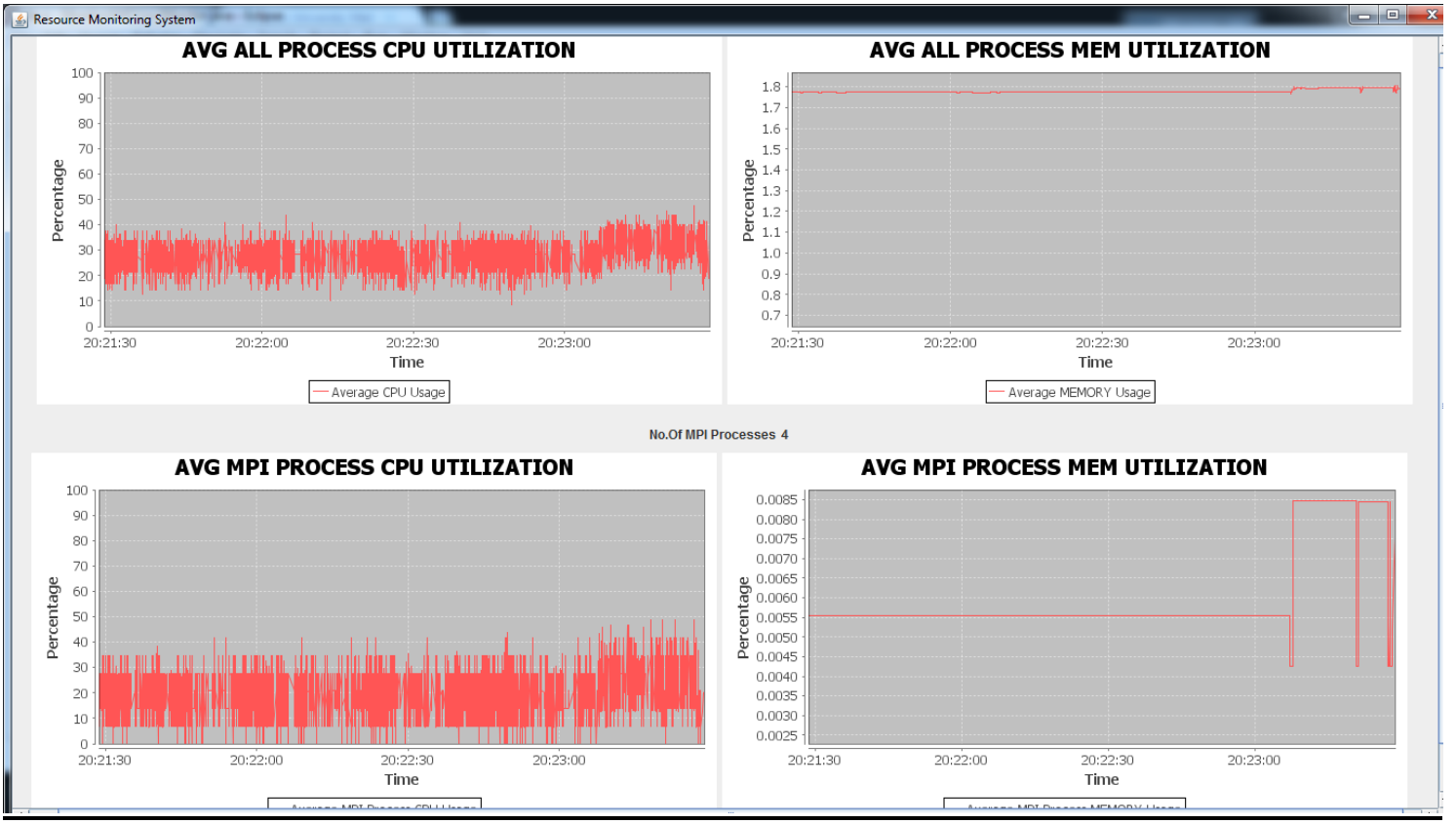
VII. ACKNOWLEDGEMENT

We'd like to thank Prof Judy Qiu, the AIs, the Futuregrid team and the members of the B534 Spring mailing list for their inputs.

REFERENCES

[1] TORQUE Resource Manager http://www.clusterresources.com/products/torque-resource-manager.php
[2] KVM Hypervisor http://www.linux-kvm.org/page/Main_Page
[3] libvirt: The virtualization API http://www.libvirt.org/
[4] Torque Qsub: http://www.clusterresources.com/torquedocs21/commands/qsub.shtml#I
[5] Torque Job submission: http://www.clusterresources.com/torquedocs/2.1jobsubmission.shtml
[6] Sergey Brin and Lawrence Page, The Anatomy of a Large-Scale Hypertextual Web Search Engine, Stanford University, WWW7 Proceedings of the seventh international conference on World Wide Web 7 , 1998
[7] http://en.wikipedia.org/wiki/Markov_chain
[8] http://en.wikipedia.org/wiki/Adjacency_matrix
[9] http://en.wikipedia.org/wiki/PageRank
[10] http://www.open-mpi.org/doc/v1.4/

Appendix 1: Dynamic Provisioning on Bare Metal



Appendix 2: Dynamic Provisioning on VM


Resource Monitoring System
AVG ALL PROCESS CPU UTILIZATION
Percentage
Time
22:01:30
22:02:00
22:02:30
22:03:00
Average CPU Usage
AVG ALL PROCESS MEM UTILIZATION
Average MEMORY Usage
No.Of MPI Processes 6
AVG MPI PROCESS CPU UTILIZATION
AVG MPI PROCESS MEM UTILIZATION