|  |
| --- |
| **B534 – Distributed Systems** |
| **Final Report** |
| By Abhimanyu Jha and Monisha Thakur – Group Number 15 |
|  |
|  |
|  |
| **12/4/2012** |

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| No | Topic | Page Number |
| 1 | **Introduction** | 3 |
| 2 | **The Projects** | 4 |
| 2.1. | The Page Rank Algorithm | 4 |
| 2.2. | MPI Page Rank | 7 |
| 2.3. | Resource Monitoring Project | 9 |
| 2.4. | Performance Analysis on Dynamic Cloud | 11 |
| 2.5. | Dynamically Switch/ Provision Clusters on Academic Cloud | 11 |
| 2.6. | Observations | 15 |
| 2.7. | Snapshots | 20 |
| 3. | **Analysis** | 22 |
| 4. | **Conclusion** | 22 |
| 5. | **Acknowledgement** | 22 |
| 6. | References | 22 |

1. **INTRODUCTION**

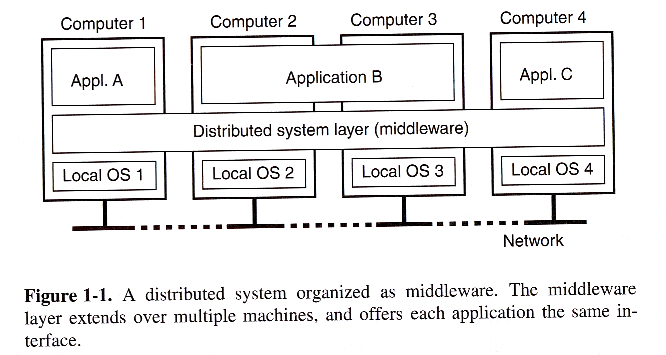
A distributed system is a piece of software that ensures that:

“A collection of independent computers that appears to its users as a single coherent system.”

Distributed computing is a field of computer science that studies distributed systems. A distributed system consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal. A computer program that runs in a distributed system is called a distributed program, and distributed programming is the process of writing such programs. Distributed computing also refers to the use of distributed systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers, which communicate with each other by message passing.

A distributed system may have a common goal, such as solving a large computational problem. Alternatively, each computer may have its own user with individual needs, and the purpose of the distributed system is to coordinate the use of shared resources or provide communication services to the users. Other typical properties of distributed systems include the following:

* The system has to tolerate failures in individual computers.
* The structure of the system (network topology, network latency, number of computers) is not known in advance, the system may consist of different kinds of computers and network links, and the system may change during the execution of a distributed program.
* Each computer has only a limited, incomplete view of the system. Each computer may know only one part of the input.



* 1. **Aim**

With the emergence of the Net as a computing platform, distributed applications are being widely deployed by organizations. Understanding the principles/theory and the technologies underlying distributed computing and systems design is increasingly important. The Internet has greatly expanded the scope and importance of distributed systems to include Web 2.0 sites, Information retrieval (search), Utility (cloud) computing, P2P systems and the Internet of things. Further science is facing an unprecedented data deluge and the emergence of data oriented analysis as a fourth paradigm of scientific methodology after theory, experiment and simulation.

This course was outlined as follows:

* Algorithms – PageRank
* Synchronization – MPI
* Processes – Virtualization
* Communication – Messaging Middleware
* Synchronization – Lamport, NTP and SNTP
* Performance Analysis – Scaling
  1. **Scope**

In this course we have developed four projects, i.e. the sequential PageRank algorithm, the Parallel PageRank algorithm (MPI PageRank), Resource Monitoring System and a Provisioning system that allows users to run and monitor parallel MPI program on Baremetal and Virtual machines which calculates the memory, CPU usage of the system as well as of the MPI program on the system.

In the Performance analysis on academic cloud, MPI PageRank computation was executed on two different environments Baremetal and Eucalyptus using the services provided by Futuregrid. Speed up on both the environments was calculated along with the analysis of the graph charts.

Dynamic provisioning provides the ability and possibility to use shared resources in a shared cloud environment. Client sends a request specifying their needs to resource manager to obtain computing resources. The requested resources are generated or instantiated dynamically.

1. **The Projects**
   1. **The Page Rank Algorithm**

PageRank uses the hyperlink structure of the web to view in-links into a page as a recommendation of that page from the author of the in-linking page. Since in-links from good pages should carry more weight than the in-links from marginal pages each webpage is assigned an appropriate rank score, which measures the importance of the page. The PageRank algorithm was formulated by Google founders Larry Page and Sergey Brin as a basis for their search engine. After webpages are retrieved by robot crawlers are indexed and cataloged, PageRank values are assigned prior to query time according to perceived importance. The importance of each page is determined by the links to that page. The importance of any page is increased by the number of sites which link to it.

Page Rank is a result of a mathematical algorithm based on web graph model where web pages are the nodes of the graph and hyperlinks are the edges in the graph. The rank value of a page indicates the importance of that page. Each hyperlink to a page counts as a vote of support for that page. The page rank of a page is recursively calculated and it depends on the number and page rank metric of all the pages that link to it. If a page is linked to by many pages that have a high page rank the rank of the page itself will be high too. However, if there are no incoming links to a page then there is no support for that page.

The Page Rank of a webpage is calculated as follows:

PR (A) = (1-d) + d [PR (P1)/N (P1) + ... + PR(Pn)/N(Pn)]

Where,

PR (A) is the Page Rank of a page A.

PR(P1) is the Page Rank of a page P1.

N (P1) is the number of outgoing links from the page P1.

d is a damping factor the range of which is 0 < d < 1 and is set to 0.85.

In the general case, the Page Rank value for any page p can be calculated as :

PR(p)=(1-d)/N+d\* ∑\_(q∈Set)▒(PR(q))/(L(q))

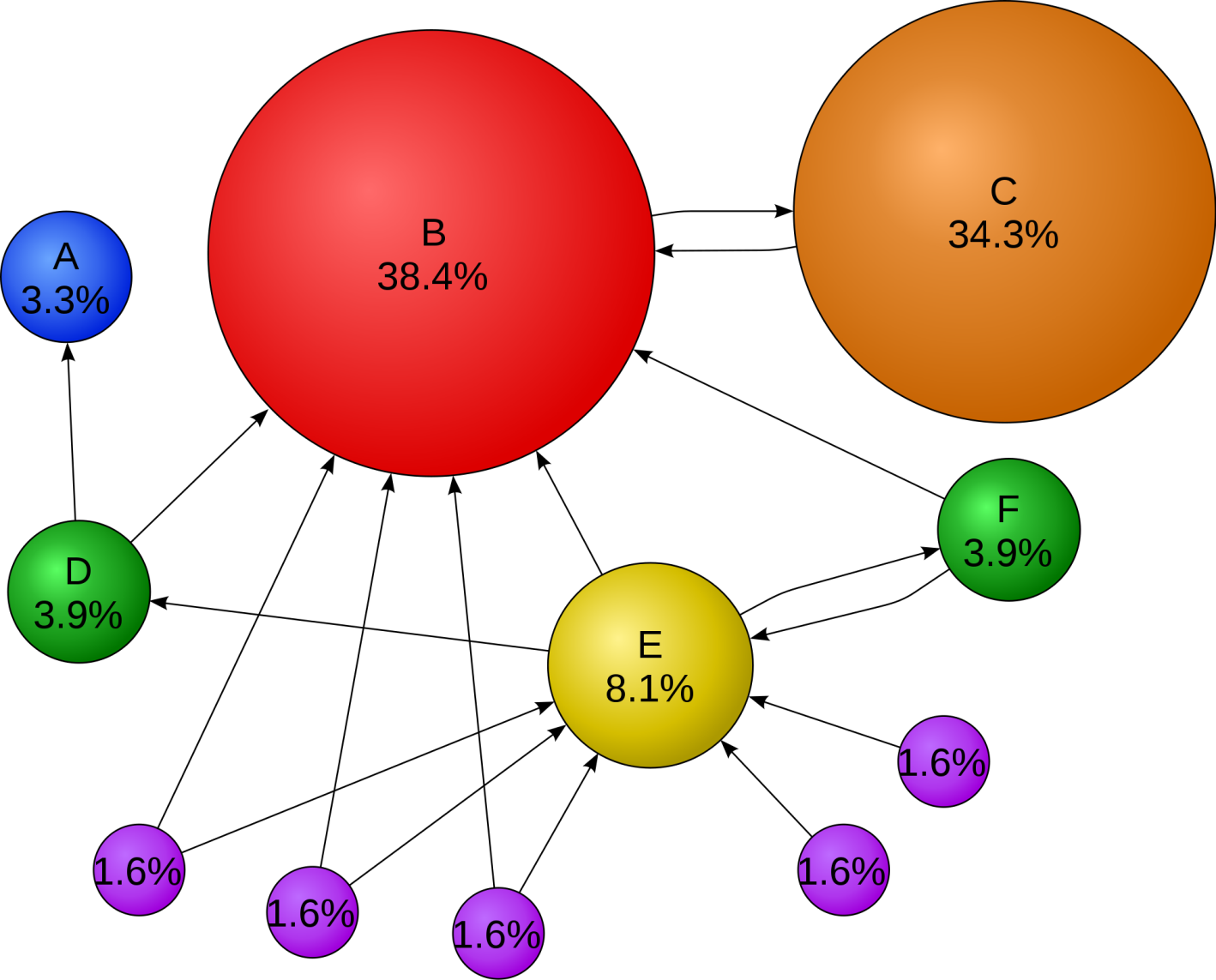
Where,

L (q) is the out degree of each web page in the vertices set

PR (q) is the Page rank of q

d is the damping factor= 0.85 generally

N is the total number of the unique URLs.



**Figure 1: Mathematical Page Rank for Simple Network**

**Flowchart for Sequential PageRank Algorithm**

**Read the command line inputs.**

**input?**

**Y**

**NO**

**Call method initial\_rank**

**Read the input and calculate initial rank**

**Compute the initial ranks**

**NO**

**Sort URLs in descending order**

**Iterations?**

**Y**

**Write top 10 URLs to output files.**

**Compute new page ranks**

**Iterations ++**

* 1. **MPI PageRank**

Developing parallel PageRank is an active research area for both in industry and academia and numerous algorithms have been proposed. The key idea in developing parallel PageRank is to partition PageRank problem into N sub problems so that N processes solve each sub-problem concurrently. One of simple approaches in partitioning is a vertex-centric approach. The graph of PageRank can be divided into groups of vertices and each group will be processed by a process. We take this approach for our MPI PageRank implementation.

The main components of the project are as follows:

The project constitutes of two classes, MPIFunctions and MPIRank.

**MPIFunctions:**

* public static void BuildMatrix (HashMap<Integer,ArrayList<Integer>> h,String input)
  + Used to build the adjacency matrix.
* public static int PageRank(HashMap<Integer,ArrayList<Integer>>Matrix,ArrayList<Integer>index,int url, int totalNumUrls, int numIterations, double threshold, double[] rank\_store, Intracomm C)
  + Uses 7 arguments; the Adjacency Matrix, and Array list, the source urls, and the total number of urls, the number of iterations, threshold value, and array\_store containing the ranks and the MPI communicator.
* public static void read (String input, ArrayList<Integer> index, HashMap<Integer,ArrayList<Integer>> Matrix,Intracomm C) throws IOException
  + division of the input file into chunks and then using send and receive functions to distribute the chunks across processes.
* public static void init\_rank(double[] values, int size)
  + used to calculate the initial ranks.

**MPIRank**

* static public void main(String[] args)
  + the main function which calls individual functions from the MPIFuntion class, and then sends over the results to a sort function in the same class.
  + MPI.Init( ) : This function initializes the MPI environment
  + MPI.Finalize ( ): This should be the last function called and as the name suggests this function finalizes the MPI environment.
* public static long sort(double[] store\_rank, String output)
  + Sorts the results of the PageRank, and writes the top 10 rank values into an output file name.

**Architecture**

**Read the command line inputs.**

N

**input?**

System.exit(-1)

**Y**

**MPI is initialized**

N

**Iterations?**

**Call MPIFunctions.read()**

**MPIFunctions.read() divides the input into chunks and calls the BuildMatrix() functions.**

**Y**

**Sort URLs in descending order**

**Each process calculate the PageRank() for the chunk it receives.**

**Write top 10 URLs to output files.**

**MPI.Finalize() is called**

MPI is the Message Passing Interface which is used to execute parallel Java applications on a network of processors. MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementers, and users. Although Java does not have an official MPI binding, several groups attempt to bridge the two, with different degrees of success and compatibility. The MPI Communicator objects connect groups of processes in the MPI session. Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology. MPI also has explicit groups, but these are mainly good for organizing and reorganizing groups of processes before another communicator is made.

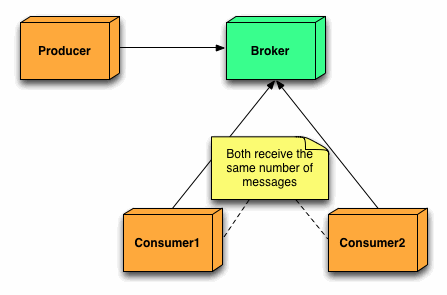
MPI includes variants of each of the reduce operations where the result is returned to all processes in the group. MPI requires that all processes participating in these operations receive identical results. The all-reduce operations can be implemented as a reduce, followed by a broadcast. However, a direct implementation can lead to better performance. The MPI.Bcast() broadcasts a message from the process with rank "root" to all other processes of the communicator.

* int **MPI\_Allreduce**(void\* sendbuf, void\* recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm)
* int **MPI\_Bcast**( void \*buffer, int count, MPI\_Datatype datatype, int root, MPI\_Comm comm )
  1. **The Resource Monitoring Project**

Resource monitoring in distributed systems is essential in order to manage the resources efficiently and to detect failures as well as to optimize the distributed application performance. We implement a system that monitors the CPU and memory utilization for a distributed set of nodes. We achieve this by having a producer that sends messages to the Broker which in turn routes these messages to the consumer and plot the CPU utilization and memory utilization percentages on that graph.

Message brokering is defined as an architectural pattern for message validation, message transformation and message routing. It acts as an intermediary thereby minimizing the awareness among the producer and consumer thereby enabling efficient message exchange between the two. One of the key things in understanding broker-based messaging is that the production, or sending of a message, is disconnected from the consumption of that message. The broker acts as an intermediary, serving to make the method by which a method is consumed as well as the route that the message has travelled orthogonal to its production. When a standard message is sent to a queue from a producer, it is sent to the broker, where the message remains in the message queue.

The architecture essentially consists of three modules- The Message Broker, Monitoring Daemon (Publisher) and a Monitoring UI (Consumer). In our project we use Narada Brokering or Apache ActiveMQ Messaging System as the broker.

****

**System Flow**

Host mpj Page Rank Algorithm on Server to run in background

Make the Publisher run on Server and compute CPU and Memory Utilization

Send Messages

Message Broker

Check queue

Consumer checks queue and plots statistics using Jfreechart.

**Class Publisher:**

The publisher runs on the server and continuously populates the queue with useful information such as CPU usage and Memory usage. The following are the functions used in the code: sigar.getProcMem, sigar.getMem, sigar.getCPUPerc and sigar.getProcCPU.

**Class Consumer:**

It interfaces with the ActiveMQ broker and runs on the local machine and captures this useful information from the queue. It calls the JfreeChart Class to represent this information in the form of a graph.

**JfreeChart:**

It is a class which presents a real-time graphical usage of cpu, memory using time series and by receiving the queued messages from activeMQ broker using consumer class.

* 1. **Performance Analysis on the Academic Cloud**

The main aim of this project was to analyze the performance of the MPI-PageRank algorithm on the Academic Cloud. For analysis, we run the program on two different modes; the Bare Metal and the Eucalyptus VMs on the FutureGrid. Having observed, marked and analyzed the time reading on these, we represent them quantitatively by plotting charts for them. The charts denote differences in these readings and help us understand the gradient. Cloud computing is the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet). The name comes from the use of a cloud-shaped symbol as an abstraction for the complex infrastructure it contains in system diagrams. Cloud computing entrusts remote services with a user's data, software and computation. Cloud computing systems provide access to large amounts of data and computational resources through a variety of interfaces. On a cloud resources can be acquired and released on-demand and that the user interface is kept fairly simple. In addition, resources provided by cloud computing systems hide a great deal of information from the user through virtualization.

The sequential and parallel PageRank programs were tested on two environments on FutureGrid namely on bare metal and eucalyptus VM’s and performance analysis was done on these environments by calculating the speed up and also various experiments were performed by changing the parameters like number of processes, number of nodes and number of cores.

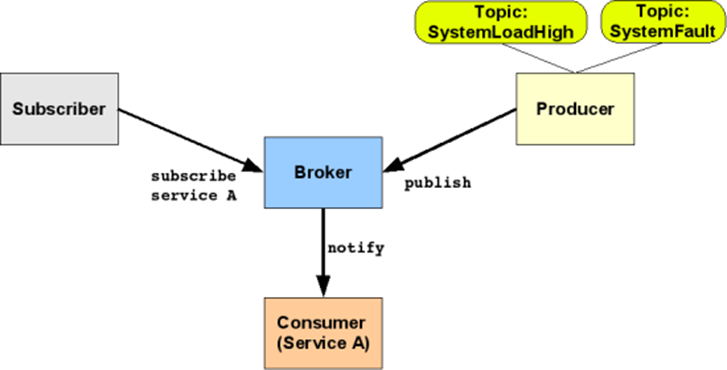
* 1. **Dynamically Switch/ Provision Clusters on Academic Cloud**

Dynamic provisioning provides the ability and possibility to use shared resources in a shared cloud environment. Client sends a request specifying their needs to resource manager to obtain computing resources. Users of FutureGrid obtain baremetal machines from Torque manager or boot up a set of virtual machine with India Eucalyptus. A dynamic provisioning system has been built for switching between baremetal and virtual machine cluster environments using XCAT, MOAB, Torque scheduler.

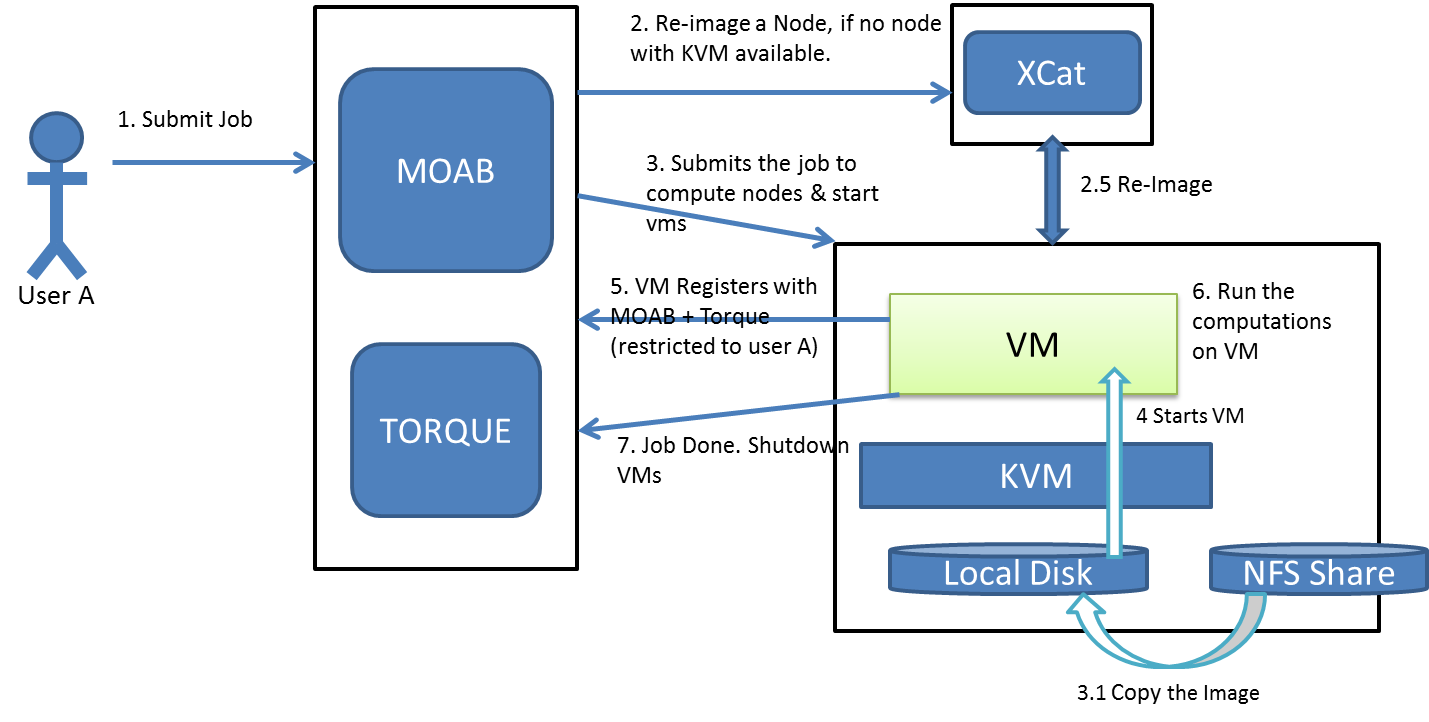
Resource monitoring in distributed systems is essential in order to manage the resources efficiently and to detect failures as well as to optimize the distributed application performance. We implement a system that monitors the CPU and memory utilization for a distributed set of nodes. We achieve this by having a producer that sends messages to the Broker which in turn routes these messages to the consumer and plot the CPU utilization and memory utilization percentages on that graph.

We use Apache Active MQ messaging systems which uses messaging system which implements the pub/sub architecture, where a node publishes its messages to a broker queue for which one or more subscribers subscribe to the queue.

Message brokering is defined as an architectural pattern for message validation, message transformation and message routing. It acts as an intermediary thereby minimizing the awareness among the producer and consumer thereby enabling efficient message exchange between the two. One of the key things in understanding broker-based messaging is that the production, or sending of a message, is disconnected from the consumption of that message. The broker acts as an intermediary, serving to make the method by which a method is consumed as well as the route that the message has travelled orthogonal to its production. When a standard message is sent to a queue from a producer, it is sent to the broker, where the message remains in the message queue.



**SYSTEM ARCHITECTURE**



In system architecture client submits a job to MOAB or TORQUE scheduler, the job is submitted to the compute nodes , VM starts and VM registers with MOAB and TORQUE, now the mpi program is executed on VM and after the computation the job is done. With batch jobs, we submit a PBS job script, which waits in queue until resource is available and then gets executed.

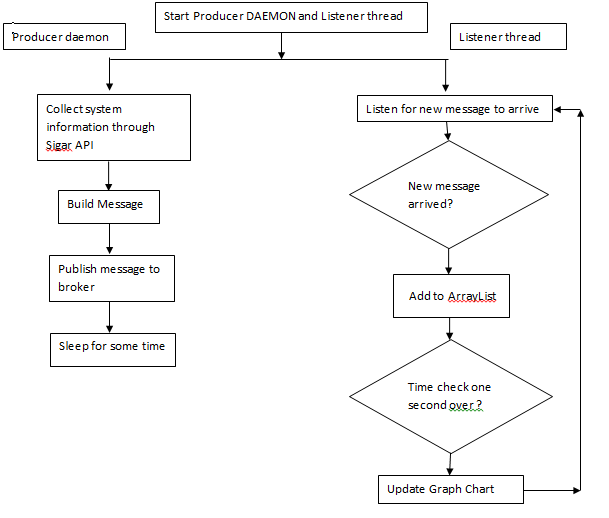
**SigarAPI**: System Information Gatherer (SIGAR) is a cross platform API for collecting software inventory data. Sigar API users are given access to system memory, network route, connection tables, file system detection etc.

**ActiveMQ**: is the most popular and powerful open source messaging and Integration Patterns server. It supports many cross language clients and protocols from Java, C, C++, and PHP. It supports java open wire transport and the Stomp JMS mapping. It is designed for high performance clustering, client server and peer based communication.

**Implementation:**

The producer is deployed on the nodes, which aggregates data from them. This data is then sent to the broker at interval of three seconds which sends it to the subscriber. The subscriber aggregates the data coming from all the nodes that it receives in a second and passes it to the JfreeChart plotter. The scripts for the bare metal and VM form an important part of the project. These scripts are used to send the jobs to the Portable Batch System (PBS). As we are supposed to invoke the producer on 2 nodes, we use ssh to send command to the other node. We then invoke the mpi program on the first node. The producer runs as long as the MPI program runs and is then shut down. In case of VM we need to SSH to both the nodes to start the Producer. Running the producer as a daemon is important. To achieve this we append it with an ‘&' so that it runs in the background. In the producer, we maintain an internal thread which monitors when the MPI process becomes active and then retrieves its PID. This is used by SIGAR to collect process specific information. As soon it exits, an exception is caught and the producer exits. The producer implements a timer object to collect information every three seconds.

**Flowchart**



* 1. **Observations**
     1. **Results for Sequential PageRank:**

|  |  |
| --- | --- |
| URL | Page Rank Value |
| 4 | 0.1332679121289077 |
| 34 | 0.1193701159498179 |
| 0 | 0.11093625335974695 |
| 20 | 0.07987998874804304 |
| 146 | 0.0629266241379715 |
| 2 | 0.04843305853104551 |
| 12 | 0.020820823691011202 |
| 14 | 0.01699919306847242 |
| 16 | 0.013151225678444194 |
| 66 | 0.01143554205924632 |

* + 1. **Results for Parallel PageRank:**

|  |  |
| --- | --- |
| URL | Page Rank Value |
| 4 | 0.1332679121289077 |
| 34 | 0.1193701159498179 |
| 0 | 0.11093625335974695 |
| 20 | 0.07987998874804304 |
| 146 | 0.0629266241379715 |
| 2 | 0.04843305853104551 |
| 12 | 0.020820823691011202 |
| 14 | 0.01699919306847242 |
| 16 | 0.013151225678444194 |
| 66 | 0.01143554205924632 |

* + 1. **Bare Metal Observations for a single worker node with the MPI-Pagerank.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| iterations | input size | delta | np | IO | TO |
| 10 | 10000 | 0.0001 | 2 | 10.08651 | 2.48325 |
|  | 20000 |  |  | 13.59015 | 5.10195 |
|  | 50000 |  |  | 18.97203 | 4.46082 |
|  | 100000 |  |  | 30.15875 | 8.50131 |
|  | 200000 |  |  | 56.80773 | 13.96038 |
|  | 500000 |  |  | 122.7087 | 32.30934 |
|  | 1000000 |  |  | 287.2714 | 75.4908 |
|  | 2000000 |  |  | 635.721 | 406.1062 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| iterations | input size | delta | np | IO | TO |
| 20 | 10000 | 0.01 | 2 | 9.85173 | 0.88494 |
|  | 20000 |  |  | 12.7323 | 3.08826 |
|  | 50000 |  |  | 19.96533 | 1.97757 |
|  | 100000 |  |  | 34.81328 | 3.00233 |
|  | 200000 |  |  | 55.72413 | 5.37285 |
|  | 500000 |  |  | 116.4509 | 9.90591 |
|  | 1000000 |  |  | 283.2711 | 26.93649 |
|  | 2000000 |  |  | 649.2119 | 60.16689 |

The results demonstrated in the above tables are the average values of 3 readings of the MPI-Page rank algorithm run on the BareMetal mode. By keeping no. of iterations constants, IO and TO values were recorded for different input sized url files.

* + 1. **Bare Metal Observations for two worker node with the MPI-PageRank.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| iterations | input size | delta | np | IO Time | TO Time |
| 10 | 10000 | 0.01 | 4 | 9.94203 | 0.82173 |
|  | 20000 |  |  | 14.48412 | 1.59831 |
|  | 50000 |  |  | 21.38304 | 2.38392 |
|  | 100000 |  |  | 42.98951 | 3.57792 |
|  | 200000 |  |  | 62.14446 | 4.99359 |
|  | 500000 |  |  | 149.4194 | 10.12263 |
|  | 1000000 |  |  | 338.9772 | 74.28981 |
|  | 2000000 |  |  | 788.626 | 169.4028 |
| iterations | input size | delta | np | IO Time | TO Time |
| 10 | 10000 | 0.01 | 10 | 11.20623 | 1.3545 |
|  | 20000 |  |  | 15.23361 | 2.21235 |
|  | 50000 |  |  | 25.09437 | 4.85814 |
|  | 100000 |  |  | 47.78182 | 7.40868 |
|  | 200000 |  |  | 70.33467 | 10.91727 |
|  | 500000 |  |  | 167.4975 | 52.69005 |
|  | 1000000 |  |  | 347.8717 | 100.9554 |
|  | 2000000 |  |  | 821.9738 | 223.1042 |

* + 1. **Eucalyptus Observations for a single worker node with the MPI-PageRank.**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| itr | input | d | np | IO | TO | itr | input | d | np | IO | TO |
| 10 | 10000 | 0.01 | 2 | 18.08709 | 1.20099 | 10 | 10000 | 0.0001 | 2 | 18.26769 | 4.55112 |
|  | 20000 |  |  | 26.41275 | 3.56685 |  | 20000 |  |  | 25.58199 | 9.1203 |
|  | 50000 |  |  | 41.25807 | 3.28692 |  | 50000 |  |  | 41.95338 | 8.3979 |
|  | 100000 |  |  | 68.6018 | 7.12875 |  | 100000 |  |  | 67.56335 | 15.83105 |
|  | 200000 |  |  | 89.44215 | 10.01427 |  | 200000 |  |  | 92.34981 | 22.80075 |
|  | 500000 |  |  | 199.8068 | 60.73578 |  | 500000 |  |  | 216.6297 | 90.24582 |
|  | 1000000 |  |  | 537.294 | 34.53975 |  | 1000000 |  |  | 512.6783 | 88.4037 |

* + 1. **Eucalyptus Observations for two worker node with the MPI-PageRank.**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| itr | input | d | np | IO | TO | itr | input | d | IO | TO |
| 10 | 10000 | 0.0001 | 8 | 23.86629 | 7.61229 | 20 | 10000 | 0.0001 | 24.43518 | 8.60559 |
|  | 20000 |  |  | 34.33206 | 17.95164 |  | 20000 |  | 34.10631 | 15.45033 |
|  | 50000 |  |  | 51.71481 | 24.73317 |  | 50000 |  | 50.63121 | 27.35187 |
|  | 100000 |  |  | 90.39906 | 55.24439 |  | 100000 |  | 95.23652 | 58.37984 |
|  | 200000 |  |  | 129.1561 | 88.39467 |  | 200000 |  | 139.5948 | 86.81442 |
|  | 500000 |  |  | 329.9562 | 183.2909 |  | 500000 |  | 310.8939 | 195.1383 |
|  | 1000000 |  |  | 713.3249 | 952.5476 |  | 1000000 |  | 720.1425 | 673.1414 |

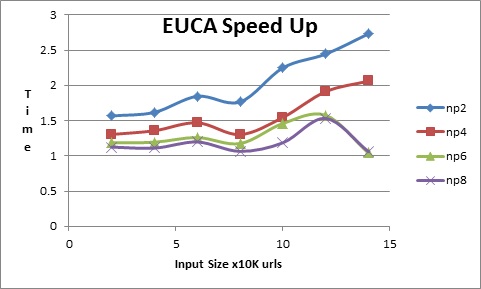
* + 1. **BareMetal VS Eucalyptus**

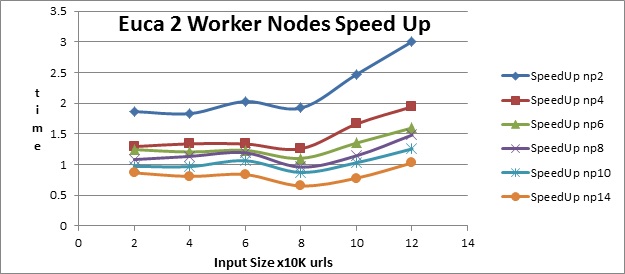
|  |  |  |
| --- | --- | --- |
| **Parameter** | **Baremetal** | **Eucalyptus** |
| Instance class type (VM) | - | M1.large |
| Worker Nodes | 2 Nodes(8 cpu/node) | 2VMs(2 cores per m1.large) |
| Datasets | 50000 , 100000 , 500000 | 50000 , 100000 , 500000 |
| Num\_of\_Processes | 2,4,8,16,32 | 2,4,8,16,32 |
| Threshhold | 0.000001 | 0.000001 |
| Iterations | 10 | 10 |

**Speed Up Readings:**

Speed Up for Eucalyptus

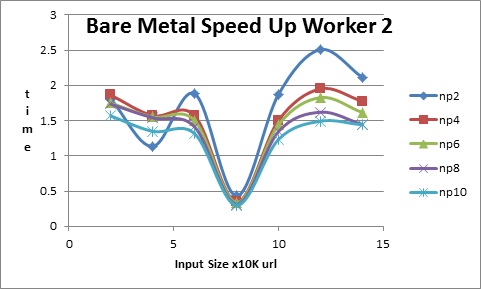
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Size | np=2 | np=4 | np=6 | np=8 |
| 10k | 1.56802 | 1.305134 | 1.18741 | 1.125039 |
| 20k | 1.618888 | 1.358382 | 1.195731 | 1.111558 |
| 50k | 1.845924 | 1.47062 | 1.260918 | 1.197821 |
| 100k | 1.766498 | 1.303346 | 1.176448 | 1.06519 |
| 200k | 2.249859 | 1.547392 | 1.46073 | 1.187852 |
| 500k | 2.444955 | 1.91221 | 1.575631 | 1.531629 |
| 1M | 2.732292 | 2.061444 | 1.040737 | 1.060477 |





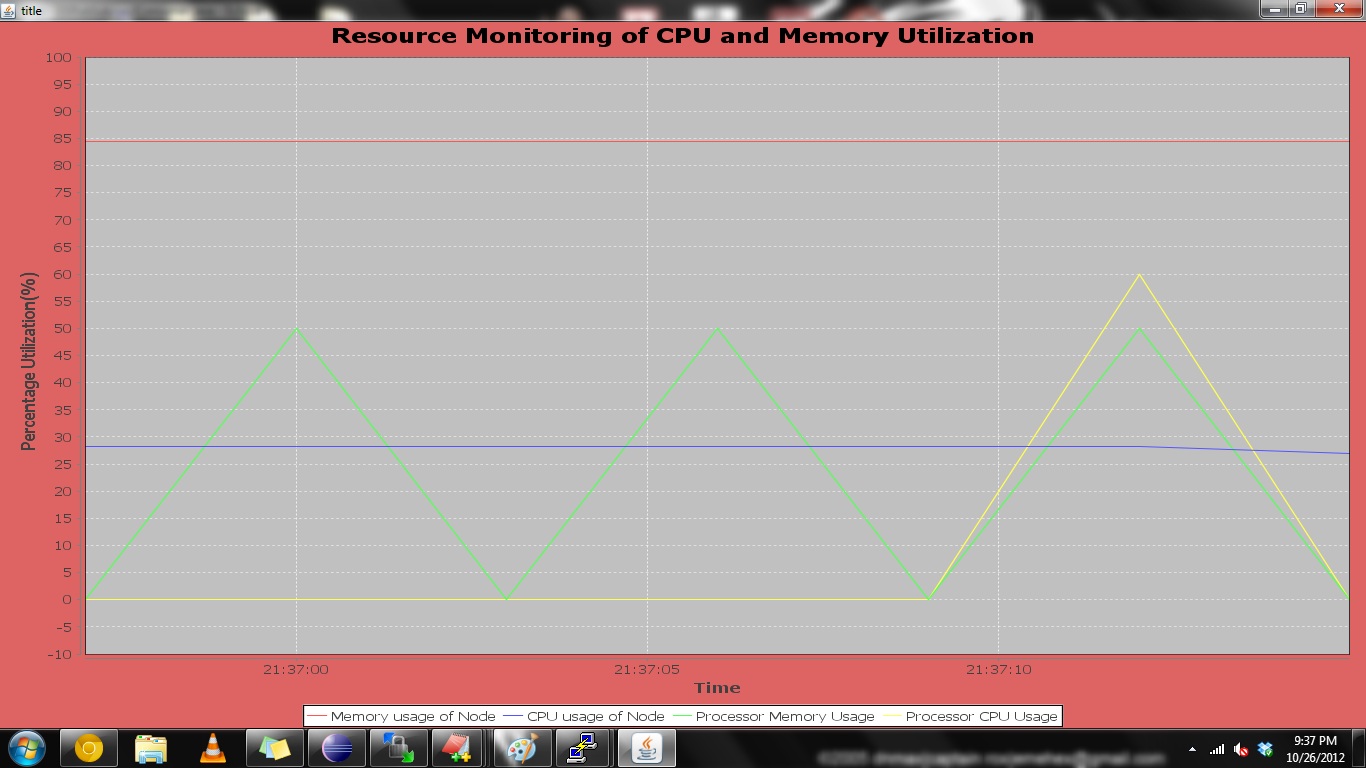
Speed Up for **BareMetal**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input Size | np=2 | np=4 | np=6 | np=8 | np=10 |
| 10k | 1.805085 | 1.862974 | 1.749487 | 1.735234 | 1.567137 |
| 20k | 1.135864 | 1.571889 | 1.553279 | 1.54065 | 1.349288 |
| 50k | 1.882873 | 1.571472 | 1.502646 | 1.412155 | 1.314815 |
| 100k | 0.432016 | 0.347478 | 0.337574 | 0.313409 | 0.298402 |
| 200k | 1.87165 | 1.503641 | 1.442588 | 1.346088 | 1.234041 |
| 500k | 2.508042 | 1.951322 | 1.828063 | 1.62128 | 1.493631 |
| 1M | 2.116079 | 1.776875 | 1.617688 | 1.446559 | 1.44121 |

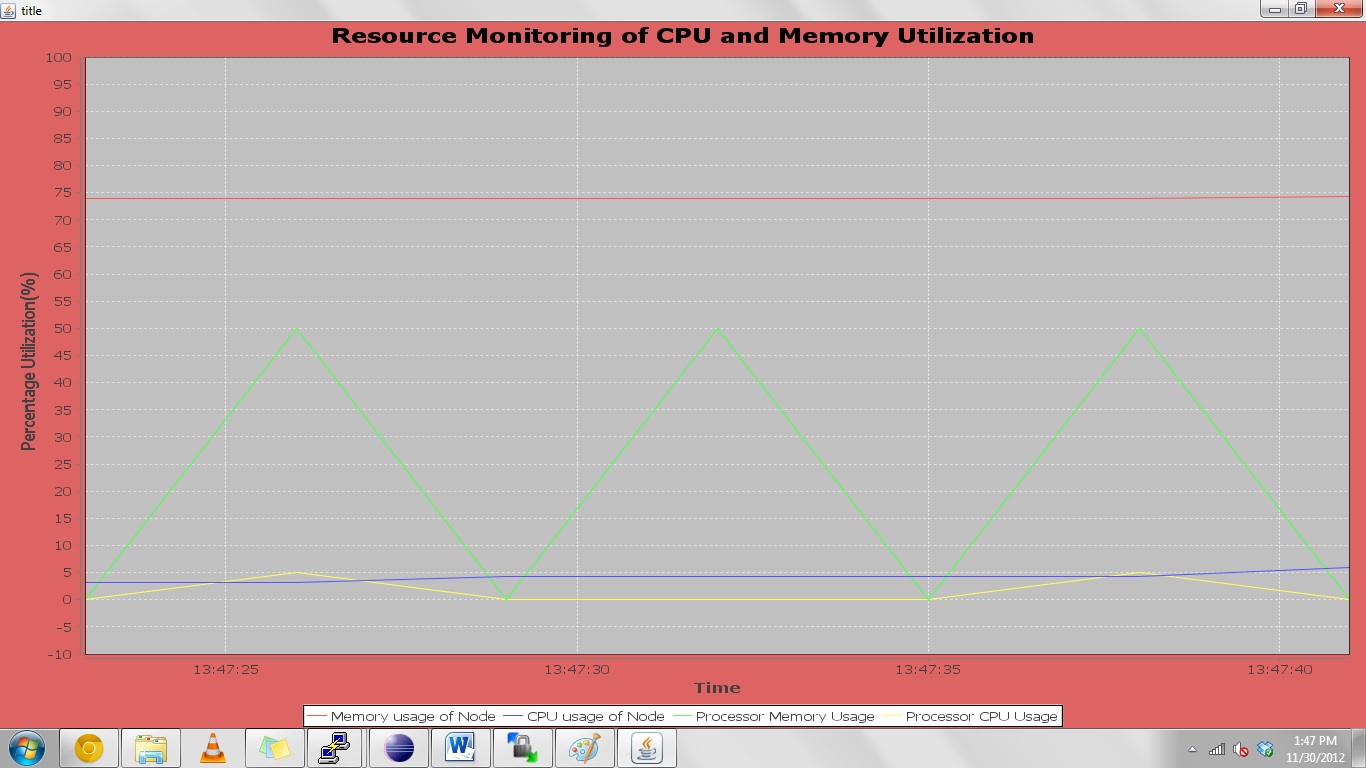


* 1. **Snapshots**

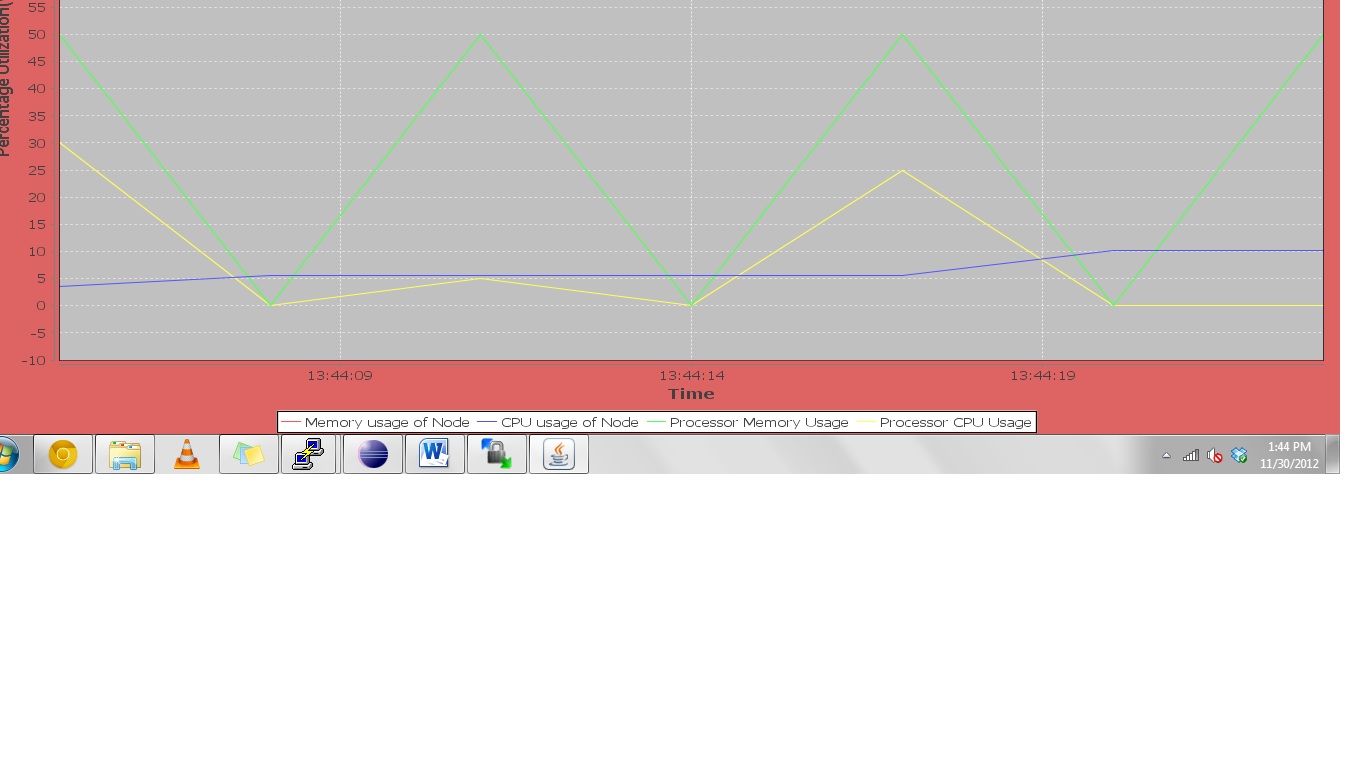
Snapshot of Resource Monitoring



**Screenshot for Bare Metal**



**Screenshot for VM**

****

1. **ANALYSIS:**
2. The parallel page rank algorithm is a lot more efficient than the sequential page rank algorithm. Thus, for large input set parallel page rank should be the choice.
3. For the resource monitoring project we made the following observations:
   1. As the value of error threshold gets lower the time to calculate page rank increases.
   2. If we increases number of MPI processes then IO time increases with the increasing number of MPI processes.
   3. If we increase the number of processes per node then page rank calculation will get faster.
   4. Here IO time may remain same at times but Job turnaround time decreases as the processing becomes more parallel.
4. We observed that the execution time in Bare Metal is lesser than that in Eucalyptus Virtual Machine. This is because the core capacity is quite large compared to core capacity of Virtual Machines.
5. We also observed that Job turn-around time is always lesser than its corresponding IO time.
6. **CONCLUSION:**

We came across a lot of hurdles while implementing these projects but they have enhanced our knowledge about a distributed system in many ways. It has been a good learning experience implementing each of these projects namely- Sequential page rank, parallel page rank, resource monitoring and dynamic switch/provision clusters on Academic Cloud which bare great importance in the real world.

1. **ACKNOWLEDGEMENTS:**

We would like to express our gratitude and are thankful to Professor Judy Qiu for her valuable assistance which helped us complete this project successfully. We would also like to thank Stephen Wu for his guidance and support throughout this project. We feel it is the combined efforts of all that helped making this project a success. We would also like to that Team Futuregrid for letting us use their resources.

**6. REFERENCES:**

[1]. Eucalyptus tutorial on Future Grid

<https://portal.futuregrid.org/tutorials/eucalyptus>

[2]. Wiki Link for Message Broker:

<http://en.wikipedia.org/wiki/Message_broker>

[3]. Sigar API Souce:

<http://www.hyperic.com/products/sigar>

<http://www.hyperic.com/support/docs/sigar/org/hyperic/sigar/SigarProxy.html>

[4]. JfreeChart Source:

<http://www.java2s.com/Code/Java/Chart/JFreeChartDynamicDataDemo.htm>

<http://stackoverflow.com/questions/5048852/using-jfreechart-to-display-recent-changes-in-a-time-series>

[5]. Publisher-Subscriber Source:

<http://stackoverflow.com/questions/3032827/lightweight-publish-subscribe-framework-in-java>

[6]. Process Id Reference:

<http://www.coderanch.com/t/109334/Linux-UNIX/UNIX-process-ID-java-program>

[7]. MPJ Linux Guide:

<http://mpj-express.org/docs/guides/linuxguide.pdf>

[8] Page Rank Wiki Page:

<http://en.wikipedia.org/wiki/PageRank>

[9] Google Page Rank:

<http://www.smashingmagazine.com/2007/06/05/google-pagerank-what-do-we-really-know-about-it/>

[10] Efficient parallel page rank algorithm:

<http://www.cs.ucsb.edu/~kmeeker/ppagerank_report.pdf>

[11] MPI Wiki Page

<http://en.wikipedia.org/wiki/Message_Passing_Interface>

[12] MPI Allreduce

<http://www.mpi-forum.org/docs/mpi-11-html/node82.html>

[13] MPI\_Bcast

<http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Bcast.html>