**B534 Project 4 Dynamically switch/provision clusters on Academic Cloud**

Ninad Faterpekar ([nfaterpe@indiana.edu](mailto:nfaterpe@indiana.edu)), Chen Jiang (jiang@indiana.edu)

**Introduction**

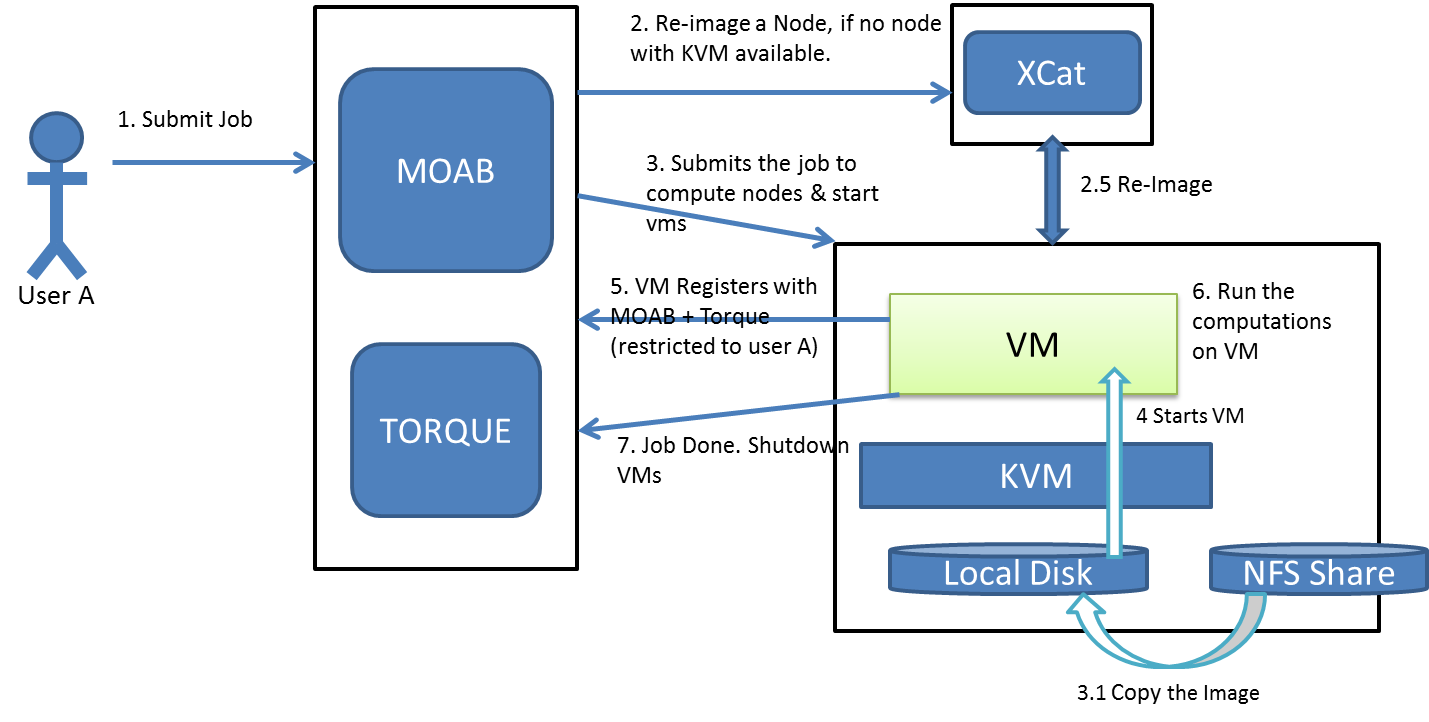
The basic idea of dynamic provisioning is that end users, mainly administrators, can use a client application in the centralized console to deploy or instantiate computing nodes or virtual machines. From this simple client application, an end user is able to easily and dynamically instantiate computing nodes or virtual machines.

The main goal of this project is to develop simple shell scripts that perform dynamic provisioning and switching as discussed above on FutureGrid. Besides, the scripts also automatically run mpipagerank program and set up computing resourcing monitoring system on each of obtained computing nodes. Then, remote users can visualize the resource usage for overall and the specific application mpipagerankfrom by using a graphic application on a desktop/laptop.

**Main Technologies and Components**

The scripts we developed employ TORQUE Resource Manager to acquire computing resources and set up proper environment. The PBS job scheduler reads the job directives specified in the scripts, and assigned and set up the computing resources accordingly. The scripts execute mpipagerank program which is detailed in Project 1. The scripts use the monitoring system from Project 3 with additional function that monitor a user specified process. We use Apache ActiveMQ as the message broker. More information about the system can be found in Project report. Fig 1 shows the architecture of the whole dynamic provisioning system

.



**Fig 1.** User interactions with Dynamic provisioning system (borrowed from project instruction document)

**Overview of Scripts**

Fig 2 shows overview of scripts “run\_pagerank\_bm” and “run\_pagerank\_vm”, performing dynamic provisioning on bare metal nodes and virtual machines, respectively. These two scripts are integrated into one single script “group02\_switch.sh”, which switches between bare metal nodes and virtual machines.

**Capturing Process Cpu and Memory Usage**

In project 3, we used getCpuPrecList() function from Sigar getting the number of CPU first, and added up all usedCpuPercent on every CPUs (multicore) together then divide by the core number to get the average overall CPU usage. Sigar directly provide a getMem().getUsedPercent() function to get the overall memory usage.

In this project, in order to capture the process CPU and Memory usage, we have to consider the scenario of multiply processes running on multi cores. Sigar actually provide a function called getMultiProcCpu and getMultiProcMem to get the total CPU and Mem usage of processes with same process name. However, getMultiProcCpu function provided by sigar 1.6.4 behaves in Solaris mode not Irix mode, which means %CPU cannot be more than 100%, if default mode is irix, when there is only one CPU, result 20% means 20% of one CPU, if there are four CPUs the total %CPU can be 400%, but sigar hardcodes if (procCpu>1.0) procCpu =0.99 and using Solaris mode as default which may cause incorrect result on multicores.

So we changed to useProcCpu.getPercent() function get single process cpu usage via the pid list we got from Process finder function in Sigar and added all together. Fortunately, ProcCpu.getPercent() lib changed to use Irix mode since 1.6.2, we don’t have to worry about the multicore %CPU issues. However, this function is getting CPU usage via the time difference which makes us have to call this function twice to get data, otherwise, a single call returns zero usage, and also, the interval between function calls should be more 1000ms (In fact, on bare metal nodes and virtual machine, the interval needs to be 2000ms). In this case, when there are 8 MPI processes running, it takes around 8 to 10 seconds to get process average CPU usage that is not acceptable.

Finally, we are using linux runtime top command to get and parse the kernel information about the process usage “top –b –n 1 | grep “ + pid[i] which is much stable and faster than sigar lib to get single process CPU usage, and we are still using getMultiProcMem to get memory usage and divide by the total memory to get the percentage.

See the discussion section about the accuracy of using either Sigar library or linux runtime to get CPU usage information

run\_pagerank\_bm

ProcCpuMem on node 1:

Sends message periodically Stops when there is no mpi\_main processes on node 1

qsub from node i136:

request nodes 1&2

ProcCpuMem on node 2:

Sends message periodically Stops when there is no mpi\_main processes on node 2

mpipagerank

mpirun on node 1

mpi\_main processes on node 1

mpi\_main processes on node 2

release node 1&2

run\_pagerank\_vm

qsub from node i136:

request nodes 1&2

start\_vms (v1 & v2)

wait\_for\_vms (v1 & v2)

shutdown\_vms

mpipagerank

mpirun on v1

mpi\_main processes on v1

mpi\_main processes on v2

ProcCpuMem on v1:

Sends message periodically Stops when there is no mpi\_main processes on v1

ProcCpuMem on v2:

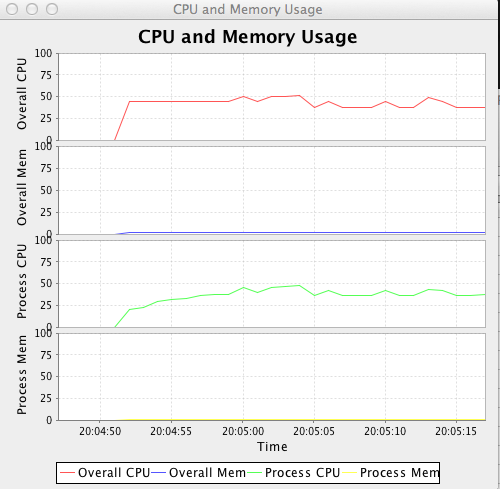
Sends message periodically Stops when there is no mpi\_main processes on v2

release node 1&2

Fig 2. Overview of scripts. The dashed arrows indicate that the starting of one process precedes the starting of other process. The solid arrow indicate that the start of one process depend on the completion of the previous process. Note: the failures of some processes may cause crash or going to other states, and these cases are not included in this figure for the sake of simplicity; this figure represents for arbitrary number of nodes.

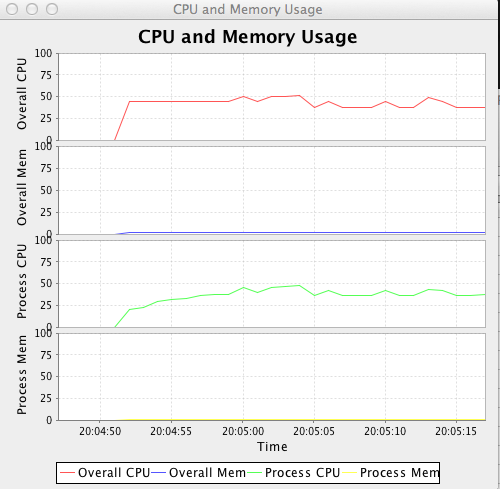
**Computing Resource Usage Monitoring System**

We ran mpipagerank program on FutureGrid using first using two bare metal nodes and then using two virtual machine instances. We set number of processes as 8, and used 1M URLs as input file. Fig 3a is a snapshot showing resource usage after the mpipagerank and the monitor daemon start on bare metal nodes. Fig 3b shows switching from using bare metal nodes to virtual machines. Fig 3c shows the resource usage when the mpipagerank program is running on virtual machines.



scac

Bqw0077q;l



**Fig 3.** Snapshots of monitoring UI. From top to bottom: overall CPU usage, overall memory usage, mpipagerank CPU usage, and mpipagerank memory usage. Overall and mpipagerank CPU usages are around 50% (on each node/virtual machine 4 out of 8 nodes are fully used), overall memory usage is around 2%, and the mpipagerank memory usage is around 1%. **a)** bare metal nodes; **b)** switching from bare metal nodes to virtual machines, the dashed line indicates the switching point. If you look at it closely, the overall CPU and memory usages drop a little after switching to virtual machines (see discussion section) ; **c)** virtual machines.

**Discussions**

*Message Synchronizatoin*

In project 3, we used three ArrayList buffers to synchronize the data from different node via timestamp information. In project 4, we also include another synchronization method, and user can switch between these two methods using args values of monitor. The new synchronization method receives all data and saves into a HashMap<MacAddress, ArrayList> according to their Mac address in a certain period like 1 second, and calculate the average based on unique Mac address first and output with the overall average. Fig 4 shows an example of calculation. In fact, we observe no significant differences between the two methods.

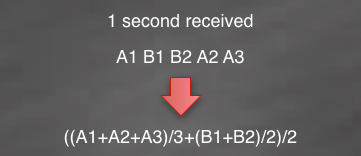


Fig 4. Calculating average resource usage in 1 second.

*VM Scripts*

We took a look at the scripts “wait\_for\_vms”, “start\_vms”, and “shutdown\_vms”. The script “start\_vms” basically writes vm IP addresses corresponding to the obtained nodes to an environment variable which is accessible by the requesting user. The script “wait\_for\_vms” checks whether each virtual machine (IP) is accessible through ssh within some timeout, and exists successfully if each vm is accessible. The script shutdown\_vms basically destroys rhels5.5 and removes relevant files.

**Conclusion:**

In this assignment, we get the chance to run our algorithm for computing pagerank in two modes: Bare metal and VMs. We tweak different parameters and record the values. Next we analyze those values to get an insight about the performance of our algorithm. We observe that communication and latency of the network plays an important role in the speedup gain. In particular, these two factors sometimes become the dominant factors such that they undermine the benefit of division of computation among processes hosted at different nodes. Also no consideration is given to balance load. However, increased input size benefit from parallelization.We also learn how to use scripts to directly run our algorithms making the process a simple one.

**Acknowledgements:**

We would like to Thank Prof. Judy Qiu for teaching the concept of Page Rank and AIsStephen Wu and Ila Jogaikar for their help and support.

# Reference

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. Resource Schedule: <https://docs.google.com/spreadsheet/ccc?key=0AtR8aHmmVF3ydGdKRk9Tb19HMVpQV25HOWctcVRoNXc#gid=0>
7. B534-Lab Notes