Exploring Distributed HPC Scheduling in MATRIX

Kiran Ramamurthy*, Ke Wang*, Ioan Raicu*†

*Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA
†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL, USA

Kramamu1@hawk.iit.edu, kwang22@hawk.iit.edu, iraicu@cs.iit.edu

Abstract - *Efficiently scheduling large number of jobs over large scale distributed systems is very critical in order to achieve high system utilization and throughput. Today's state-of-the-art job schedulers mostly follow a centralized architecture that is master/slave architecture. The problem with this architecture is that it cannot scale efficiently upto even petascales and is always vulnerable to single point of failure. This is over come by the distributed job management system called MATRIX (MAny-Task computing execution fabRIc at eXascale) which adopts a work stealing algorithm which aims at load balancing throughout the distributed system. The MATRIX currently supports Many Task Computing (MTC) workloads. This project aims at extending MATRIX in order to support the High Performance Computing (HPC) workloads. The HPC workloads are nothing but long jobs which needs multiple nodes/cores to run the tasks. It is a challenge to support HPC on the framework which supports MTC jobs. The framework is focused at efficiently scheduling sub-second jobs on available workers. The design of scheduling HPC jobs should be efficient enough in order to not hamper the efficient working of MTC tasks.*

I. Introduction

High-Performance Computing (HPC) is a distributed paradigm defined to address challenges in running large jobs spanning across multiple nodes along with the small jobs (MTC) by maintaining load balancing as well as efficient scheduling mechanism. Running HPC jobs should not be a bottleneck for the performance of MTC jobs.

Efficiently scheduling large number of jobs over large scale distributed systems is very critical in order to achieve high system utilization and throughput. Workflow systems such as Swift, have been shown to generate massive amounts of MTC tasks on Grids, Supercomputers, and clouds [16, 17]. Today's state-of-the-art job schedulers mostly follow a centralized architecture that is master/slave architecture. The problem with this architecture is that it cannot scale efficiently up-to even petascales and is always vulnerable to single point of failure. Supporting large jobs and small jobs together will affect the performance with a centralized scheduler. This is over come by the distributed job management.

MATRIX already implements work stealing algorithm in order to maintain load balancing. A new paradigm called resource stealing is introduced for the large jobs. The work stealing and resource stealing co-exists and performs better at large scales. The idea of resource stealing is to steal compute units (in these case, cores) from the neighbors in order to run a large job. The neighbors are selected in random in the similar way as work stealing. This can be improved since it has lot of alternatives to choose from. The current initial system implements random neighbor selection to steal resources. The report presents the architecture of MATRIX supporting HPC, resource stealing and the design consideration along with the results obtained so far.

MATRIX being a MTC job execution framework, the motive was to make MATRIX support HPC jobs. This needs to be done along with the co-existing work stealing mechanism. Developing an HPC support should not be a bottleneck for the MTC tasks. Making work stealing and resource stealing work as separate entities poses a lot of challenges like deadlocks, improper load balancing and executing tasks in a generic way.

II. Background Information

Many Task Computing: Many-Task Computing is a paradigm which bridges the gap between High Performance Computing (HPC) and High Throughput Computing (HTC). The MTC workloads are finer granularity tasks which takes many computing resources in order to complete many task in lesser time. The throughput is measured in terms of seconds. Tasks can be small or large, uniprocessor or multiprocessor, compute intensive or data intensive. The set of tasks may be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. The aggregate number of tasks, quantity of computing, and volumes of data may be extremely large [2][3][4][5].

High Performance computing: The HPC jobs are the ones which might needs a whole node consisting of multiple cores or more than one node to complete the tasks[4]. Current MATRIX system does not support the HPC jobs and needs to be extended in order to support the same.

Centralized Architecture: Most of the state-of-the-art job management systems today have a centralized architecture that is jobs are submitted to a single scheduler which has multiple compute nodes under it. The scheduler is responsible for submitting jobs to the compute nodes and also the load balancing. This is an inefficient as it cannot scale enough. At the same time this architecture is vulnerable for a single point of failure that is if the scheduler fails then the entire system falls apart as there would be no track of the jobs submitted as well as no scheduler to accept new jobs. Some of the examples are Condor, Slurm, Falkon [1].

Many-Task computing execution fabRIC at eXascale: MATRIX is a state-of-the-art distributed job management system which eliminates the concept of master/slave architecture. This job management system has many compute nodes connected in the cluster. Each compute node has servers running on it. A client generated jobs and submits to one of the compute nodes. Each compute node would have a scheduler. The idle nodes perform a Work Stealing mechanism in order to steal jobs from heavily loaded system. The current implementation selects the nodes randomly to steal the jobs. If the randomly selected node does not have any job or is not overloaded, the node which made the request waits for sometime before performing the work stealing again [1].

III. Proposed Work

This project aims to extend MATRIX with HPC support along with the current MTC support. This is achieved through the random approach where in the requested number of nodes is selected in random. The major considerations of the project would also be to facilitate inter process communication since a single process might be running across multiple nodes and the task dependency, resource deallocation during system under high utilization.

The proposed work is to implement a real system which supports HPC tasks on the MTC task execution framework, MATRIX. The system includes numerous compute nodes to execute submitted jobs and client to generate the jobs.

The work proposed is to run HPC jobs on MATRIX. The system takes the information as to how many cores each job needs to run. Resource stealing is implemented in order to obtain the resources to run the job. The nodes are selected in random to steal resources. The jobs to be executed are the sleep jobs. In case of high system utilization, the resources need to be released if enough resources cannot be obtained. In order to avoid deadlocks and starvation, back-off time will be implemented as required.

a. Architecture

The MATRIX-HPC has 4 components, a client, scheduler, worker and a ZHT server. The client is a benchmarking tool used to generate tasks to submit to the scheduler. The scheduler places the tasks on the wait queue of the worker. The worker takes the tasks from the ready queue and executes it. The ZHT server is used to maintain the metadata of the tasks like Task ID, which is a combination of the client id and self index of the worker.

MATRIX-HPC supports single core jobs, referred to as MTC tasks and multiple core jobs, referred to as HPC tasks. The HPC task does not support dependency between each task, while the dependency is maintained within the task.

The Client initializes the workload of given type and submits the workload to one or more compute nodes. With the help of ZHT, the task dispatcher could submit tasks to one arbitrary node or to all the nodes in a balanced distribution [11]. In the background, the compute nodes distribute the task among themselves with 2 mechanisms, work stealing and resource stealing. The ZHT server maintains information of the all the tasks distributed across compute nodes. The ZHT also keeps information about the sub tasks of an HPC tasks as to which node the sub tasks has been migrated to. Whenever the task needs to be migrated or broken into sub tasks and needs to be migrated, the ZHT is updated with the task id and description.

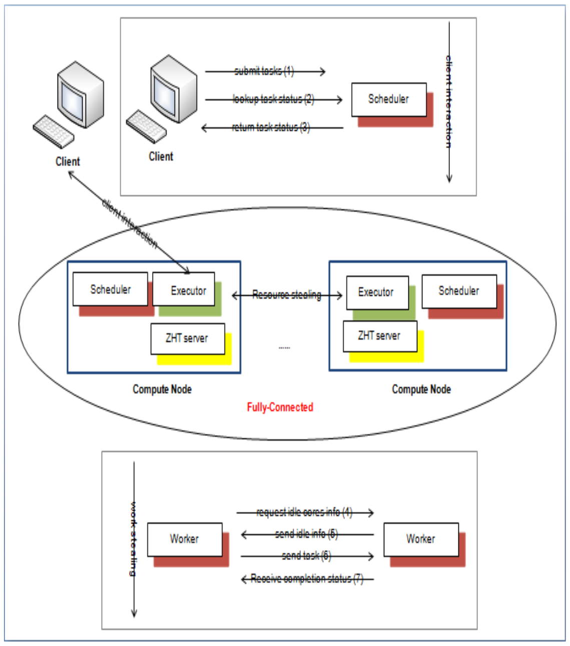


Figure 1: MATRIX-HPC architecture

b. Implementation

Figure 2 shows the step-by-step process of the execution of MATRIX-HPC.

The client submits the tasks to the scheduler. The task ID and description is stored in ZHT and NoVHT. The task is then placed in the wait queue. Once the task is ready to execute, that is there is no more dependency (in case of MTC tasks), it is moved to the ready queue.

The worker picks the task from the ready queue. The ZHT server is looked up using the task ID to get the description. Once the description is received, the information regarding the number of cores required for executing and the source index is retrieved.

If the number of cores required is 1 and the source index is -2, the task is an MTC task. The index -2 is used to identify that the task is not migrated from other node as a HPC subtask. If the number of cores required is more than 1, a check is performed if the task can be run on a single machine or needs resource stealing. If the task can be performed on the same node, the task waits until it acquires the required number of resources and executes the tasks. If the index is -1, then the source is current node and the task is placed in the complete queue. If the index is 0 or more, then the result is sent back to the source with retrieved index. If the number of cores is greater than 1and sufficient resources are not available on the same node, resource stealing is initiated.

i. Resource Stealing

Random neighbor Selection: Every worker has a membership table is aware of all the other workers. The neighbors can be selected in 2 ways for resource stealing. One is static that is the neighbor from which the resource needs to be stolen has to be pre-defined. In dynamic selection multiple nodes are selected in random from the membership table to look up for the resource. Once the neighbors are chosen, the resource information is requested from the selected neighbors. The number of neighbors to be selected can be set. At present the square root of the total number of nodes available would be selected. For example if there are 1024 nodes, then for each resource stealing initiation, 10 nodes will be selected to run the tasks [10].

Migrating Tasks: After selecting the neighbors in random, ZHT server is used to get the resource information. There are 2 types of resources, number of cores idle with the worker and the worker's ready queue. Each worker replies with the number of cores idle with them and the size of their ready queue.

The information received is stored in a structure array which consists of the neighbor's index and the number of idle cores associated with it. Once receiving information from all the selected nodes, a check is performed to evaluate if enough resources are available on all the nodes in order to migrate the task. If enough resources are available, then task is broken into sub tasks and migrated to the selected nodes. For example if task 1 needs 10 cores to execute the task on a cluster with 4 nodes and each node having 8 cores in total, the random node selection algorithm chooses 2 nodes in random with index 1 and 2. Node with index 1 replies with the number of idle cores available to be 6 and the node with index 2 replies with number of idle cores available to be 4, the task is broken into 2 parts. One with id appended with index 1 and the number of cores information in the package updated to 6 and the other with id appended with index 2 and the number of cores information in the package updated to 4.

After breaking the task, the new task id and the description is inserted into ZHT and NoVHT keeping the main task id and description intact. The tasks are then migrated to the selected nodes and inserted at the front of the ready queue such that the tasks will the next immediate one to be executed. There are 2 design considerations in stealing the resources:

1. When requesting for resource information, none of the resources on the other node are locked. Instead only the information is collected and the task is migrated if enough resources are available. The advantage of this approach is it is not prone to deadlocks. But the major disadvantage of this approach is, 2 nodes might select same random nodes for resource stealing. Though the information received is accurate, by the time the task is placed on the ready queue, the resource would be gone. This leads to extra waiting time for the task to get hold of the resources again and execute.
2. When requesting for resource information, variable idle core information is locked along with the ready queue. This is to ensure that the information received will remain the same even after migrating the task. This is the best approach. But the drawback of this design is deadlock. Multiple parameters need to be taken into consideration while developing this method. Work stealing and resource stealing should be deadlock free when they work together. At the same time in resource stealing the chances of a circular deadlock are high. For example, 2 nodes select same random nodes with index 0 and 3, the node 1 tries to steal resource from index 0 and hence locks the resources on 0. Node 2 tries to steal resources from 3 and locks resources on 3. Now node 1 has 0, but blocks on 3 while node 2 has 3 and blocks on 0. This leads to a circular deadlock.

ALGORITHM 1. Dynamic Multi-Random Neighbor Selection (DYN-MUL-SEL)

Input: Node id (*node_id*), number of neighbors (*num_neigh*), and number of nodes (*num_node*), and the node array (*nodes*).
Output: A collection of neighbors (*neigh*).

```
bool selected[num_node];
for each i in 0 to num_node do
    selected[i] = FALSE;
end
selected[node_id] = TRUE;
Node neigh[num_neigh];
index = −1;
for each i in 0 to num_neigh−1 do
    repeat
        index = Random( ) % num_node;
    until !selected[index];
    selected[index] = TRUE;
    neigh[i] = nodes[index];
end
return neigh;
```

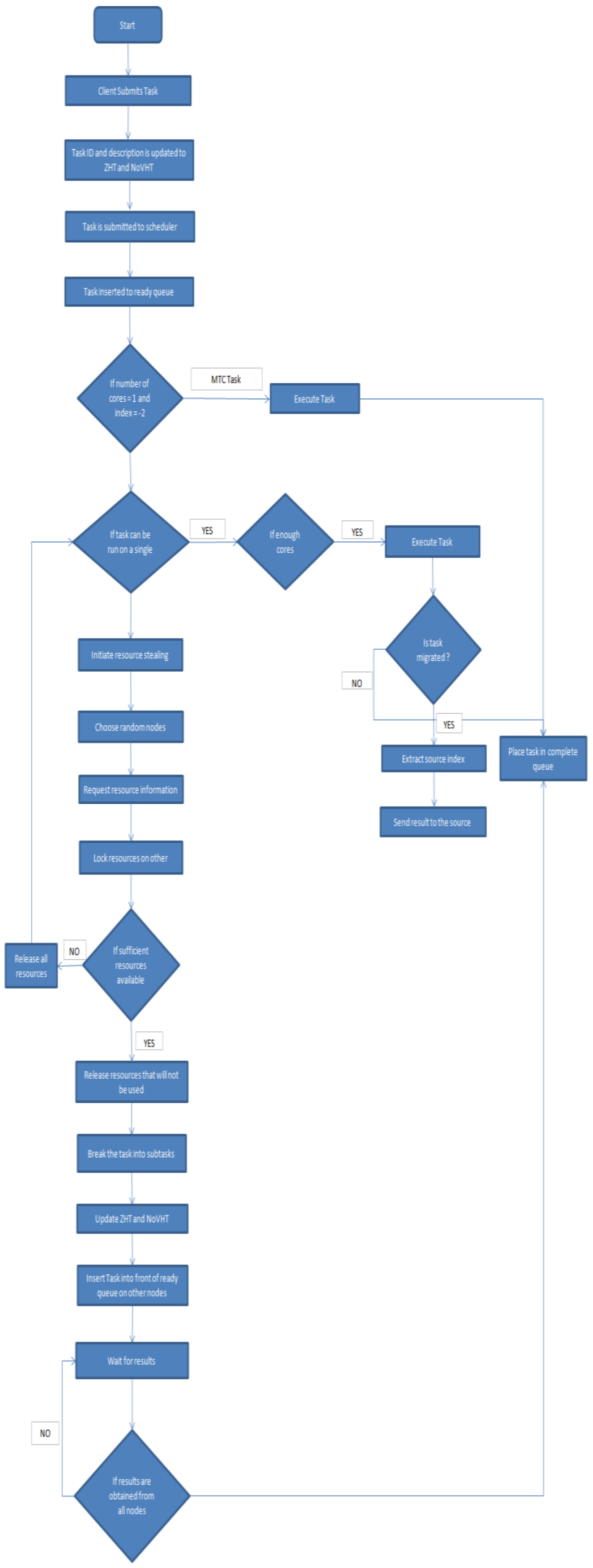

Start
Client Submits Task
Task ID and description is updated to ZHT and NoVHT
Task is submitted to scheduler
Task inserted to ready queue
If number of cores = 1 and index = -2
MTC Task
Execute Task
If task can be run on a single
YES
If enough cores
YES
Execute Task
Is task migrated ?
NO
YES
Extract source index
Send result to the source
Place task in complete queue
Initiate resource stealing
Choose random nodes
Request resource information
Lock resources on other
Release all resources
NO
If sufficient resources available
YES
Release resources that will not be used
Break the task into subtasks
Update ZHT and NoVHT
Insert Task into front of ready queue on other nodes
Wait for results
NO
If results are obtained from all nodes

Figure 2 - HPC execution sequence

```
3.   ALGORITHM 2.   RESOURCE STEALING ALGORITHM
4.   Input: Structure array with nodes selected in random,
     package of the task to be broken into subtasks.
5.   Output: NULL
6.   get_idle_core_information(idle_core_info
7.   Success=check_for_sufficient_cores(idle_core_info,nu
     m_of_cores,selected_neigh)
8.   If(!success)
9.   release_resources(idle_core_info)
10.  else
11.              for(i=0;i<selected_neigh_count)
12.
                 package=build_package_with_self_index()
13.  Update_ZHT_and_NoVHT(package)
14.
                 Migrate_Tasks(selected_neigh[i].index
15.
```

ii. ZHT and NoVoHT Updates

The ZHT and NoVHT are the integral part of MATRIX [9]. The task metadata and description is stored in the ZHT. Each worker has a ZHT server and a global NoVHT store.

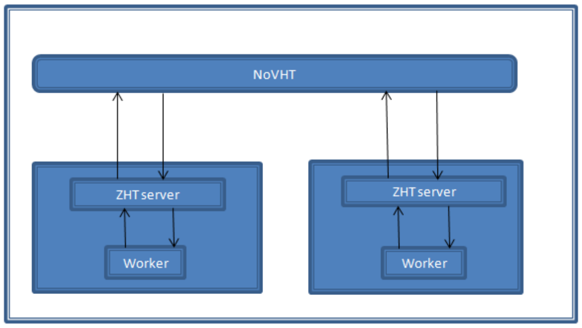


Figure 3 - ZHT and NoVHT: The relation between the worker, ZHT server and NoVHT store.

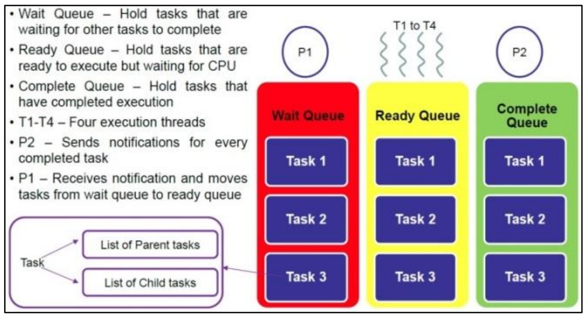
The ZHT and NoVHT is updated during the following scenarios

The client requests the task launcher to generate tasks. Once the tasks are generated, the client submits it to the schedulers or a single scheduler. The new task id and description is stored in ZHT and NoVHT, which is then used by the worker to look up the description to run the tasks. In the second scenario, when and idle node performs work stealing, the heavily loaded node while migrating tasks updates the ZHT server. The third scenario is during resource stealing. The task is looked up in the ZHT server and after stealing resources, the task is broken into subtasks with unique id's consisting of the index number of the target nodes in the task id. The package information is also updated with the number of resources each sub task needs. This information is again updated on to ZHT server before migration. Now it works in the same way on the other node which looks up for id and executes the task.

Using ZHT and NoVHT might be a bottleneck with the network performance as the package needs to be built very often and updates to the ZHT and NoVHT is made very frequently with the HPC tasks.

iii. Execution Unit

As figure 4 describes, MATRIX uses 3 queues, Wait Queue, Ready Queue and the Complete Queue. In HPC implementation, Wait Queue is not taken into consideration as the only transaction will be with the Ready Queue and the Complete Queue. The tasks are inserted at the front of the Ready Queue during resource stealing and the completed tasks are inserted back to the complete queue similar to MTC task operation. The complete queue will always have the source task id and not the sub task ids.



- Wait Queue – Hold tasks that are waiting for other tasks to complete
- Ready Queue – Hold tasks that are ready to execute but waiting for CPU
- Complete Queue – Hold tasks that have completed execution
- T1-T4 – Four execution threads
- P2 – Sends notifications for every completed task
- P1 – Receives notification and moves tasks from wait queue to ready queue

Figure 4 - Different queues used in MATRIX framework

iv. Back-off Implementation

In order to avoid deadlocks, there should be a back-off mechanism implemented to release resources when none of them can acquire it. There are 3 types of back-off implementations,

1. When the task can run on a single system that is if the number of cores needed are available on the same node, the task waits until the resources are relieved by other worker threads. This can be either done by continuous polling or make the thread sleep for sometime before it checks the resource status again.
2. When the resource stealing is initiated, the resource information is received from all the nodes. The source node then performs an evaluation if the required number of resources is available. If available, tasks are

migrated. If not, then the resources locked on all resources are relieved and the thread sleeps for a back-off time which is usually (index*1000) ms before trying again.

3. During resources stealing, the circular deadlock explained needs to be handled. This would be the challenging of all. This can be done in 2 ways. A separate thread is created to get resource information. If the result is not received for a specified amount of time, the thread would be destroyed and back-off is implemented before trying again. In the second method, while locking the resources on the other nodes, a timed lock can be implemented. In the concept of timed lock, if the lock is not obtained for the specified amount of time, the thread gives up on the mutex and returns with a non-zero number. After getting this status information, a back-off is implemented before trying again.

v. HPC Task Execution

The HPC tasks are long tasks which needs multiple nodes to run. Hence there should be a mechanism to make it start at the same time and end at the same time. Due to network latency, starting at the same time might not be possible. Hence MATRIX-HPC ensures that the tasks start at the same time on the respective nodes with barrier implementation.

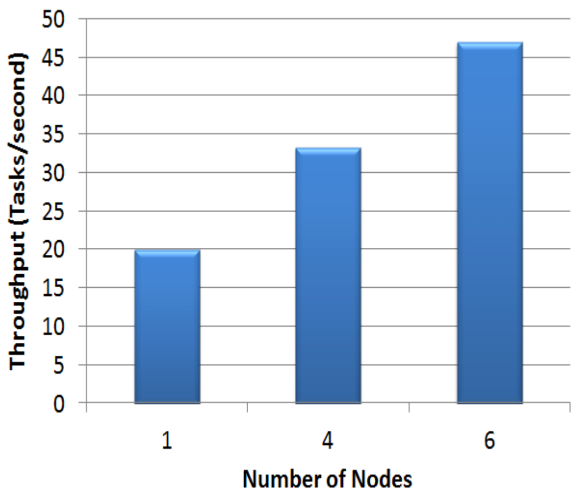
In order for the tasks to end at the same time, each sub task migrated to other nodes for execution contains the source index to which the result needs to be sent in the task description. In the meantime the source node maintains a map which has the task id as the key and the amount of task executed as the value. Each node after execution of the task, updates the map on the source node with the amount of task it ran. The source node keeps polling the map and once the map is completely updated by all the nodes, the task will be placed in the complete queue.

This implementation ensures the HPC tasks start at the same time and end at the same time across all the nodes.

IV. Evaluation

In this section, the result of MATRIX run on 6 nodes is presented. The jobs are sleep 0. A single client submits 1000 task, with each task needing 10 cores to run. This ensures the tasks spans atleast across 2 nodes. All the tests are run on Jarvis cluster. Each node in Jarvis consists of 8 cores. Jarvis has 10 nodes in the cluster. The tests were carried out till 6 nodes. That is 1, 4 and 6 nodes. The metric measured is throughput.

Throughput measures how fast the system can execute tasks. It is calculated as the total tasks executed divided by the time taken to execute all tasks. In the current HPC implementation, the execution time is measured at each server. The start time is noted at the client and the end time at the worker executing tasks.



While testing on a single node, the number of cores each task needs is 4. As we observe, the throughput for 1000 tasks is around 19~20 tasks per second. For 4 nodes and 6 nodes the number of tasks executed is 1000 with each task needing 10 cores. With 4 nodes, around 33~34 tasks can be executed per second while at 6 nodes around 47~48 tasks can be executed per second.

MATRIX-HPC was run on Amazon AWS up-to 16 nodes and a throughput of around 66~67 tasks per second were observed. The number of tasks submitted was 1000. Since the instance type used was m1.medium, each node has 1 core. Hence each task requests for 4 cores which is equivalent to 4 workers. Since the results are not concrete and cannot be made fair comparison with the Jarvis environment, it is not included in the graph above.

The trend shows that MATRIX-HPC is performing better in comparison with SLURM. But this needs to be tested at higher scales to make a fair comparison. The future works of MATRIX-HPC includes scaling the framework up-to 64 nodes and compare it with SLURM and CloudKon-HPC.

V. Related Work

The job schedulers could be centralized, where a single dispatcher manages the job submission, and job execution state updates; or hierarchical, where several dispatchers are organized in a tree-based topology; or distributed, where each computing node maintains its own job execution framework [1].

The University of Wisconsin developed one of the earliest job schedulers, Condor [6], to harness the unused CPU cycles on workstations for long-running batch jobs. Slurm [7][8] is a resource manager designed for Linux clusters of all sizes. It allocates exclusive and/or non-exclusive access to resources to users for some duration of time so they can perform work, and provides a framework for starting, executing, and monitoring work on a set of allocated nodes.

In 2007, a light-weight task execution framework, called Falkon [6] was developed. Falkon also has a centralized architecture, and although it scaled and performed magnitude orders better than the state of the art, its centralized architecture will not even scale to petascale systems. A hierarchical implementation of Falkon was shown to scale to a petascale system in, the approach taken by Falkon suffered from poor load balancing under failures or unpredictable task execution times.

A decentralized job scheduling system called Sparrow: Scalable Scheduling for Sub-Second Parallel Jobs [10] was developed by University of California, Berkeley. This job management system has many schedulers and workers. The schedulers incorporate power of 2 approaches, where in the scheduler selects 2 workers randomly to run the task. The selected workers reply with the size of the job queue. Then the scheduler places the job on the worker with lesser queue length. The drawback of this system is if 2 nodes are selected, one with 2 jobs which needs approximately 50ms each to execute and another node has only one job which needs approximately 300ms to execute, since the queue length is only one in the latter node, the scheduler places the job on it. Hence the wait time is 200ms more compared to the first one.

CloudKon now supports HPC jobs. In this version of CloudKon, the HPC jobs are placed on the SQS queue. Each worker takes a task from the HPC queue and runs the task [15]. CloudKon-HPC cannot run on any other cluster apart from Amazon AWS infrastructure since it uses Amazon web services. A fully functioning MATRIX-HPC will not have any dependency on the executing platform.

VI. Conclusion

Running MTC and HPC jobs on distributed platform poses significant challenge. The purpose of executing tasks across the nodes is to obtain a better throughput and efficiency. To achieve this, we need to design an efficient scheduler that not only works well with sub-second tasks but also with long tasks that needs to run on multiple nodes.

This project helped in understanding different state-of-the-art distributed job scheduling frameworks and also implementing one. Working alone on a real system implementation helped in pushing the limits and getting a working system in place. The project also gave an opportunity to think of different solutions that can be implemented and choose the right one.

Considering the bottlenecks in the base MATRIX system, the goal was to implement resource stealing and launching tasks on the other nodes for the current term. But the MATRIX-HPC is now a fully working system which can run HPC tasks on multiple nodes without any bottleneck. This is evaluated through running the system up-to 6 nodes. The project stayed on schedule and HPC is now implemented. A few code changes needs to be made to make it clean and work as expected.

Our future work includes:

1. The short term goal is to get the system running up-to 64 nodes scales on Amazon-AWS platform and compare it with SLURM and CloudKon-HPC.
2. The system needs to run on all platforms like Kodiak and Bluegene/P clusters with the same efficiency and performance.
3. Random node selection needs to be changed and an efficient mechanism needs to be in place to make better selection of nodes to migrate tasks. For example ZHT can be used to differentiate free workers and busy workers.
4. Integrate HPC with new MATRIX that will be built independent of ZHT.

VII. References

[1] A. Rajendran, I. Raicu. “MATRIX: Many-Task Computing Execution Fabric for Extreme Scales”, Department of Computer Science, Illinois Institute of Technology, MS Thesis, 2013

[2] I. Raicu, I. Foster, Y. Zhao. "Many-Task Computing for Grids and Supercomputers", Invited Paper, IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08), 2008, co-located with IEEE/ACM Supercomputing 2008.

[3] I. Raicu, I. Foster, M. Wilde, Z. Zhang, Y. Zhao, A. Szalay, P. Beckman, K. Iskra, P. Little, C. Moretti, A. Chaudhary, D. Thain. "Middleware Support for Many-Task Computing", Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010

[4] I. Raicu. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", Computer Science Department, University of Chicago, Doctorate Dissertation, March 2009

[5] I. Raicu, I. Foster, Y. Zhao, A. Szalay, P. Little, C. M. Moretti, A. Chaudhary, D. Thain. "Towards Data Intensive Many-Task Computing", book chapter in "Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management", IGI Global Publishers, 2009

[6] D. Thain, T. Tannenbaum, M. Livny, “Distributed Computing in Practice: The Condor Experience “Concurrency and Computation: Practice and Experience 17 (2-4), pp. 323-356, 2005.

[7] M. A. Jette, A. B. Yoo, M. Grondona, “SLURM : Simple Linux Utility for Resource Management”, 9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003), pages 44-60, Seattle, Washington, USA, June 24, 2003

[8] M. Jette and D. Auble, “SLURM: Resource Management from the simple to the Sophisticated”, Lawrence Livermore National Laboratory, SLURM User Group Meeting, October 2010.

[9] I. Raicu, Y. Zhao, C. Dumitrescu , I. Foster and M. Wilde , “Falkon: a Fast and Light-weight tasK executiON framework”, IEEE/ACM SuperComputing 2007.

[10] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica, Sparrow: Distributed, Low Latency Scheduling”, SOSP conference, 2013

[11] I. Sadooghi, I. Raicu, “CloudKon: a Cloud enabled Distributed tasK executiON framework”, Illinois Institute of Technology, Department of Computer Science, PhD Oral Qualifier, 2013

[12] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu. “ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table”, IEEE International Parallel & Distributed Processing Symposium (IPDPS) 2013

[13] K. Wang, K. Brandstatter, I. Raicu. “SimMatrix: Simulator for MAny-Task computing execution fabRIc at eXascales”, ACM HPC 2013

[14] K. Wang, A. Rajendran, X. Zhou, I. Sadooghi, K. Ramamurthy and I. Raicu, “MATRIX: MAny-Task computing execution fabRIc at eXascale”, under review at IEEE/ACM CCGrid 2014

[15] I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belgodu, P. Purandare, K. Ramamurthy, K. Wang, I. Raicu “CloudKon: a Cloud enabled Distributed tasK executiON framework”, under review at IEEE/ACM CCGrid 2014

[16] Y. Zhao, I. Raicu, S. Lu, X. Fei. "Opportunities and Challenges in Running Scientific Workflows on the Cloud", IEEE International Conference on Network-based Distributed Computing and Knowledge Discovery (CyberC) 2011

[17] M. Wilde, I. Raicu, A. Espinosa, Z. Zhang, B. Clifford, M. Hategan, K. Iskra, P. Beckman, I. Foster. "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers", Poster Presentation, Scientific Discovery through Advanced Computing Conference (SciDAC09) 2009