Extending CloudKon to Support HPC Job Scheduling

Isha Kapur, Karthik Belgodu, Pankaj Purandare, Iman Sadooghi, Ioan Raicu

Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA

ikapur@hawk.iit.edu, kbelgodu@hawk.iit.edu, ppuranda@hawk.iit.edu, isadoogh@iit.edu, iraicu@cs.iit.edu

Abstract* — With the ongoing trends in computing and data systems, the time is not far when we will have exascale systems with millions of nodes and threads of execution being the major driving force for most of the large scale, distributed operations. Today, job management systems need to support a variety of applications, such as Many-Task Computing - MTC and; High-Performance Computing - HPC, that have a bulk of tasks with finer granularity due to such high parallelism supported. Most of the popular systems today, have Master/Slave architectures where a centralized server is responsible for all resource management and job executions, making it a single point of failure and also inefficient to scale at petascale systems that require finer granular workloads. The goal of this project is to enhance the recently developed CloudKon system to support HPC jobs using the various public cloud services (Amazon's SQS, DynamoDB and EC2), and distinguishing HPC and MTC tasks by the number of cores/nodes a task requires. This would work in cohesion with the actual motive behind developing CloudKon, which is to act as a distributed job management system that can support millions of tasks from multiple users delivering over 2X the performance compared to other state-of-the-art systems in terms of throughput.

Keywords - CloudKon, Many-Task Computing, High-Performance Computing, distributed scheduling

1. INTRODUCTION

Modern job schedulers aim to efficiently distribute jobs to the various available computing resources to gain peak resource utilization and maximum throughput. With the continuous increase in the scales of the systems, there is a clear need for highly efficient job schedulers. Predictions are that by the end of this decade, we will have exascale system with millions of nodes and billions of threads of execution. These schedulers are required to be highly fault tolerant and have high scalability to handle exascale systems with millions of nodes and billions of threads of execution [21].

However paradigms like client server architecture are still being used in the state of the art systems where a centralized server is responsible for submitting the jobs to various available resources and collecting the execution results. The centralized dispatcher in these systems suffers scalability and reliability issues. With decentralization, the required scalability can be achieved.

We propose a highly decentralized and distributed system that makes use of Amazon's infrastructure services like Elastic Cloud Compute, Simple Queuing Service and the distributed NoSQL key/value store (DynamoDB). Our goal is to show that HPC is possible on the "Cloud" by using these systems and the quality of service provided is at par and sometimes better than some of the state of the art systems that run on grids and clusters.

Our code base is relatively less compared to other systems. We use multiple distributed queues to deliver tasks to the workers and NoSQL database for various purposes like duplicate check and for deadlock avoidance and detection. This is very different from other approaches like random sampling, resource stealing or the hierarchical system. This systematic approach gives our system better predictability which is useful for debugging, performance improvements and adding new features.

CloudKon[13] is the runtime system for workflow engines [22, 23, 24] that allows efficient remote execution of tasks on distributed systems. The components of this system mainly consist of a client component that submits the job, a highly distributed queue that holds the jobs and a set of nodes that poll the queue and get the job that will be executed. All these individual components are loosely coupled which abstracts the way in which the job is executed.

This report makes the following contributions:

1. Describes the modifications made to CloudKon architecture and explain the design choices made to achieve maximum throughput to support HPC jobs.
2. Explains the working of the system and how individual components are used.
3. Performance evaluations on scales from 16 to 1200 processes running on 300 nodes and comparing CloudKon with Slurm[12] and Slurm++. It outperforms these scheduling systems in terms of a high throughput.

2. BACKGROUND

This section covers necessary background information of CloudKon, Amazon's SQS, EC2, and DynamoDB, Many-Task Computing; and High Performance Computing.

2.1. CloudKon

It is a distributed task execution framework built upon cloud computing building blocks such as Amazon's SQS, EC2 and, DynamoDB. It has been designed to support

MTC applications which will be extended to HPC applications too. CloudKon offers properties like scalability for high throughput with larger scales through distributed services, load balancing at large scale with heterogeneous workloads, light-weight to have minimum overhead while working at fine granular workloads and the last one being loosely coupled which makes the system compact and robust [1].

2.2. Amazon SQS

Amazon SQS is a fast, reliable, scalable fully managed distributed message delivery fabric which can queue unlimited number of short messages. SQS makes it simple and cost-effective to decouple the components of cloud application. SQS can transmit any volume of data, at any level of throughput, without losing messages or requiring other services to be always available. The maximum size for a message is 256 KB where each 64KB chunk of payload is billed as one request or messages in a batch of ten. Messages can be sent and read simultaneously on SQS. When a worker receives a message, before removing that message, SQS locks the message in the queue which keeps other computers from processing the message simultaneously. If the message processing fails, the lock will expire and the message will be available again. SQS guarantees delivery of each message at least once, and provides highly concurrent access to messages. That also means it does not guarantee the exactly once delivery. Hence, there could be multiple copies of the same message available to read by different workers [3].

2.3. Amazon EC2

Amazon Elastic Compute Cloud is a Infrastructure as a Service (IaaS) that provides a raw infrastructure and the associated middleware to host a web service that allows anyone to run their applications on Amazon's computing resources by letting customers rent the computing resources by hour.

Amazon EC2 instance is a running virtual machine on Amazon's cloud platform to which clients are given access for using their computing resources. Each of these instances is deployed with an Amazon Machine Image with pre-configured OS and some bundled application software. Amazon offers different types of instances, each having a different compute capacity, memory size, I/O performance and storage [4]. These instances are categorized into three classes as follows:

- Reserved instances – Amazon allows us to pay upfront for each instance that we want to use during a given period of time, and in turn, gives us a lower hourly cost for each of them.
- On demand instances – You only pay for what you use, allowing easy allocation and deallocation of resources, depending on your workload requirements.
- Spot instances – These instances are more appropriate for running short-term applications under certain conditions. Amazon allows us to bid on unused EC2 capacity and run instances until the current spot instance price exceeds our bid to achieve a better utilization of their infrastructure. The spot price is set by Amazon based on the available capacity and load of their systems and it is updated in a 5 minute period.

2.4. Amazon DynamoDB

DynamoDB is a fast, fully distributed highly scalable, NoSQL database service that provides users to store and retrieve any amount of data, and serve any level of request traffic. It is able to handle large amounts of simultaneous write and read which are atomic. DynamoDB does not provide complex data access queries. It lets users save and access the data using its coordinating key [5].

2.5. Many-Task Computing

It aims to bridge the gap between the two computing paradigms High Performance Computing (HPC) and High Throughput Computing (HTC). MTC emphasizes one using many computing resources over short period of time to complete many tasks where primary metrics are measured in seconds. Tasks can be small or large, single processor 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 [6, 18, 19, 20, 25, 27].

2.6. High Performance Computing

High-performance computing (HPC) evolved to meet the increasing demands for processing speed. HPC brings together several technologies such as computer architecture, algorithms, programs and electronics, and system software under a single canopy to solve advanced problems effectively and quickly. A highly efficient HPC system requires a high-bandwidth, low-latency network to connect multiple nodes and clusters [7].

2.7. Google Protocol Buffer

Google Protocol buffer are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. We define how the data should be structured once and then use special code generator that helps us to create a data structure to easily write and read our structured data in Java. The metadata we need for job processing is available in this buffer and are fed into the various Amazon SQS instances.

2.8. Java Remote Method Invocation

This is a Java API that performs the object-oriented equivalent of remote procedure calls (RPC), with support for direct transfer of serialized Java objects. The various components are:

- Client is the process that is invoking a method on the remote object.
- Server is the process that owns the remote object.

- Object Registry: Name Server that associates the objects with names.

A server, upon starting, registers its objects with a textual name with the object registry. A client, before performing invoking a remote method, must first contact the object registry to obtain access to the remote object[17].

3. DESIGN AND PROPOSED SOLUTION

The goal of this project is to extend CloudKon, which is a distributed task management system, to support HPC tasks along with MTC tasks through various cloud services provided by Amazon like SQS, EC2 and DynamoDB.

HPC involves a lot of workers and hence requires inter-worker communications and synchronization. As a result, it slows down the system to some extent when compared to other MTC systems because of inherent latency in the network and also because of the synchronization between all the workers. One of our major design goals was to minimize the impact of our code on the existing performance of CloudKon for MTC.

The project mainly concentrates on the development of a prototype application which will extend the existing CloudKon, which supports only MTC tasks, to add HPC support. The application is tuned to run on Amazon cloud using Amazon Web Services like Amazon SQS, Amazon EC2 and Amazon EC2. The test bed mainly consists of the following Amazon Services:

1. Two SQS instances. One is for maintaining the list of incoming job submissions to the system called Global Request Queue. The second is for the sub workers to connect to the worker manager depending on the number of threads needed for each task.
2. An instance of DynamoDB that hold the number of HPC tasks that are being currently run on the system.
3. EC2 M1.Medium instances of varied scales.

As stated before our work was to add on top of the existing CloudKon which was developed for ground up by Iman Sadooghi in the DataSys lab at Illinois Institute of Technology. Our design includes the following components that are added to the CloudKon Architecture.

1. SQS to hold the HPC messages. If a HPC job requires n workers, n messages will be put into this queue. We had the idea of using DynamoDB but the main disadvantages are that it becomes comparatively to run and it will not let us traverse the whole
2. Modification of the client component to include the number of workers required for each HPC jobs. This is a critical step because we decide if the job is HPC or MTC depending on this value stored in each message.
3. A secondary DynamoDB that holds the state of system like number of HPC jobs that have been picked up and are in the system, waiting for more workers and the total number of workers that are needed to fulfill these jobs.
4. Various timeout mechanisms for deadlock detection and recovery.

3.1. Archicture Overview

This section explains about the various components that we are using and interaction between them. We concentrate on the components that we added by us to enable HPC support for CloudKon. We also discuss the modification made to existing components to get critical information regarding the HPC job requirements.

Client: The working of the client is inherited from the original CloudKon architecture. The Client component is independent of other parts of the system. It can start running and submitting tasks without the need to register itself into the system. The client component is multithreaded and can submit tasks in parallel. For efficient message passing and minimizing the communication cost, we use message bundling to group ten tasks together before submitting it to the system.

Global Request Queue (GRQ): An instance of SQS that is the main pool of tasks. The clients submit the jobs/tasks to this pool and the worker picks up these tasks for execution. This is highly scalable and any number of clients can put tasks into the queue without having to register or obtaining authorization. We rely on the access keys provided by Amazon for authorization. The maximum size of each message in SQS is 256KB which is sufficient for our implementation.

Client Response Queues (CRQ): Before submitting a job to the Global Request Queue each client will create a queue for himself and this queue will be the output stream to the workers. After the execution of a job, the workers will write the output data to corresponding client request queues.

Google Protocol Buffer (GPB): This buffer is used to store metadata like Number of workers required to execute a task, nature of the job - in our case it is a sleep job; and Client Response Queue IDs for the workers to submit their results to.

Workers: These are the EC2 instances that perform the actual task execution. They poll the GRQ to get the task that they have to execute. These workers can join and leave the system any time during the execution. This makes our system dependent on the GRQ for scalability and will not put any additional loads on the workers like registration and authorization. The workers are subdivided into two categories and this is applicable only to HPC jobs:

- Worker Managers: The worker who polled the GRQ and got a HPC job.
- Sub Workers: The workers who help the manager to execute the HPC job.

Any of the workers can be either a sub worker or worker manager depending on the messages in the HPCQ and GRQ.

HPC Queue (HPCQ): The queue that holds the messages for the sub workers to be picked up. The manager puts messages into this queue for other workers to pick up and execute the HPC job. Only the workers have access to this queue and it is used to hold HPC job metadata like

- Worker Manager IP and Port number for RMI communication.
- Message Pickup count is used to differentiate between valid and stale messages.

HPC DynamoDB: A record in DynamoDB helps us to make informed decisions with respect to the number of HPC jobs in the system and the number of workers that are needed to fulfill these HPC jobs. This instance is queried every time a new HPC job is picked up. The values in stored here help us decide if the sub worker requirement can be satisfied by the system. This is configured for a high read write throughput so that it can handle a lot of queries by different workers. Only the worker manger has access to DynamoDB and updates the values once it starts executing the job and after the job is executed.

3.2. Workflows in CloudKon-CKHPC:

3.2.1. Client

When client starts, it will first create a response queue for himself and then it will submit all jobs. Once the jobs are submitted, client will poll client response queue (CRQ) to check whether any response has been sent by worker or not. If it finds any response in the CRQ, it will retrieve the response i.e. message and the same will be deleted. Once client gets responses for all the jobs he submitted, then it will delete the respective CRQ.

3.2.2. Worker

Each worker in the worker pool will first check HPC worker manager queue (HPCQ). If there are any messages, implying that they would have been put by some other worker, it will pick one message and start working as a sub-worker. If the HPCQ is empty, it will check the Global Request Queue (GRQ). If it contains any job to be executed, get job/s from this queue and run as a worker manager. If this queue is also empty, repeat the same procedure until we reach to the limit of checking empty queues. There is a strict rule that the workers have to check the HPCQ before the GRQ. Hence the jobs that are already in the system waiting for sub workers will be executed first. This helps us in creating architecture with systematic execution framework for HPC jobs requiring different sub workers.

3.2.3. Worker Manager

When worker picks up job/s from the GRQ, it will first identify whether the current job is MTC or HPC. If the job picked up is MTC, then worker will start executing this job on his own and after completion send the response back to the client using the respective CRQ.

If the job is HPC, then worker manager will consult DynamoDB, to check whether the worker requirements for this HPC job are satisfied or not. This will help to avoid deadlocks in the system. If HPC job worker requirements cannot be satisfied by the system, then the worker will leave this job and start searching for new jobs.

If worker manager gets a green signal from the DynamoDB, then worker manager will put n-1 messages, where n is required number of workers, into the HPCQ, from where other workers willing to work on some jobs, will pick these messages and communicate with this worker manager to share his work. Worker manager will wait until he gets all the required number of workers. Once he gets all the workers, he will send task notifications to every other sub-worker waiting for task from this worker manager. Once task is sent to all sub-workers, worker manager itself

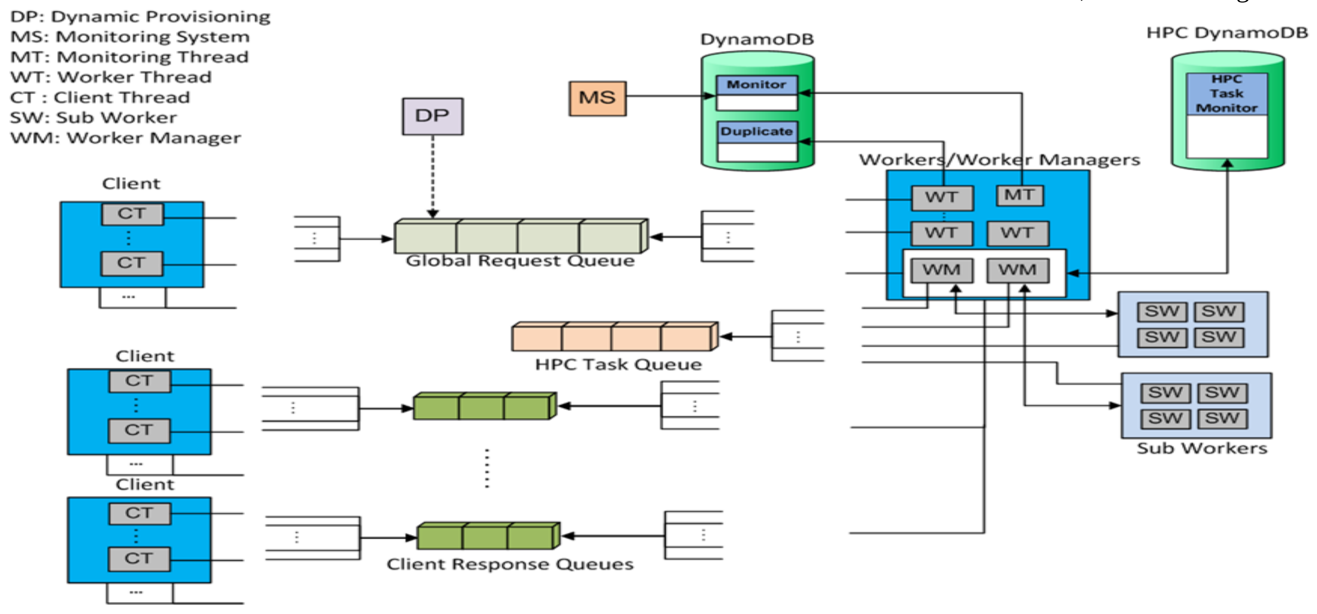

Figure 1: Architecture of CKHPC

will execute his own task and then wait for responses from the associated sub-workers. Once worker manager gets all the responses, he will send the response back to the client to its respective CRQ.

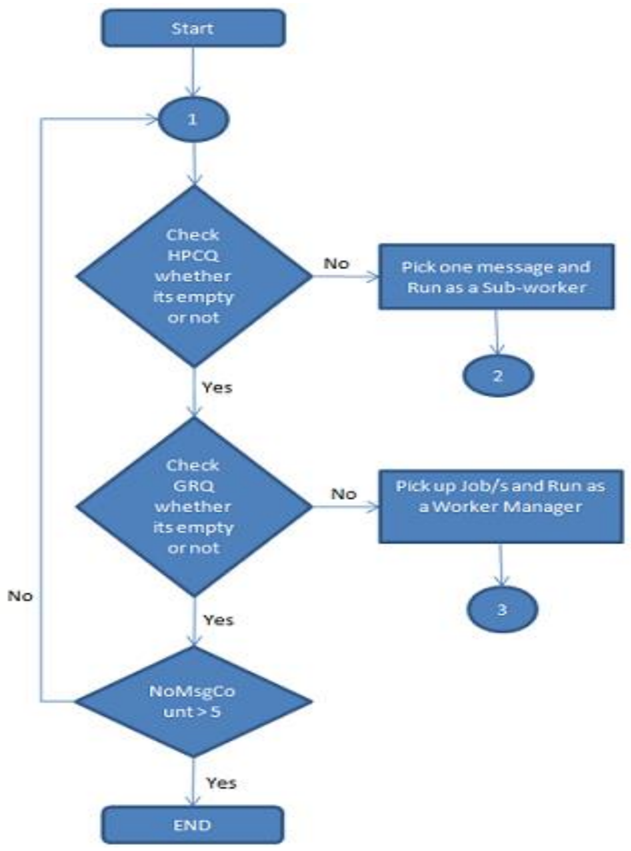
Start
1
Check HPCQ whether its empty or not
No
Pick one message and Run as a Sub-worker
2
Yes
Check GRQ whether its empty or not
No
Pick up Job/s and Run as a Worker Manager
3
No
Yes
NoMsgCount > 5
Yes
END

Figure 2: Main worker flow chart

Deadlock Avoidance and Deadlock Detection: Whenever any worker manager gets a HPC job containing ‘n’ tasks, it will first check the DynamoDB, to find whether its worker requirements will be satisfied or not. The HPC DynamoDB contains information about how many workers are currently behaving as worker manager, each working on HPC jobs. It will also contain the information about how many workers are needed to satisfy the current HPC workload taken by worker managers. With this available information, a worker manager willing to work on a HPC job, will check these values and decide, whether to keep working on this HPC job or not and depending on it, will update the values in DynamoDB. In this way, we are limiting the number of workers from becoming worker managers and avoiding deadlocks in the system, arising when all workers are either associated with some worker manager waiting for task from them or all worker managers are still waiting for few more sub-workers.

When a worker manager waits for all the required workers, it will also start the timer. When this timer expires, the worker manager will go and check the DynamoDB values, to see if a deadlock has occurred in the system. If the deadlock has not occurred, then he will wait for all sub-workers. If at all a deadlock occurs, which is a rare case, then the worker manager will notify all the associated sub-workers to leave this task and along with itself and will release all the resources, thereby solving the issue of deadlock recovery.

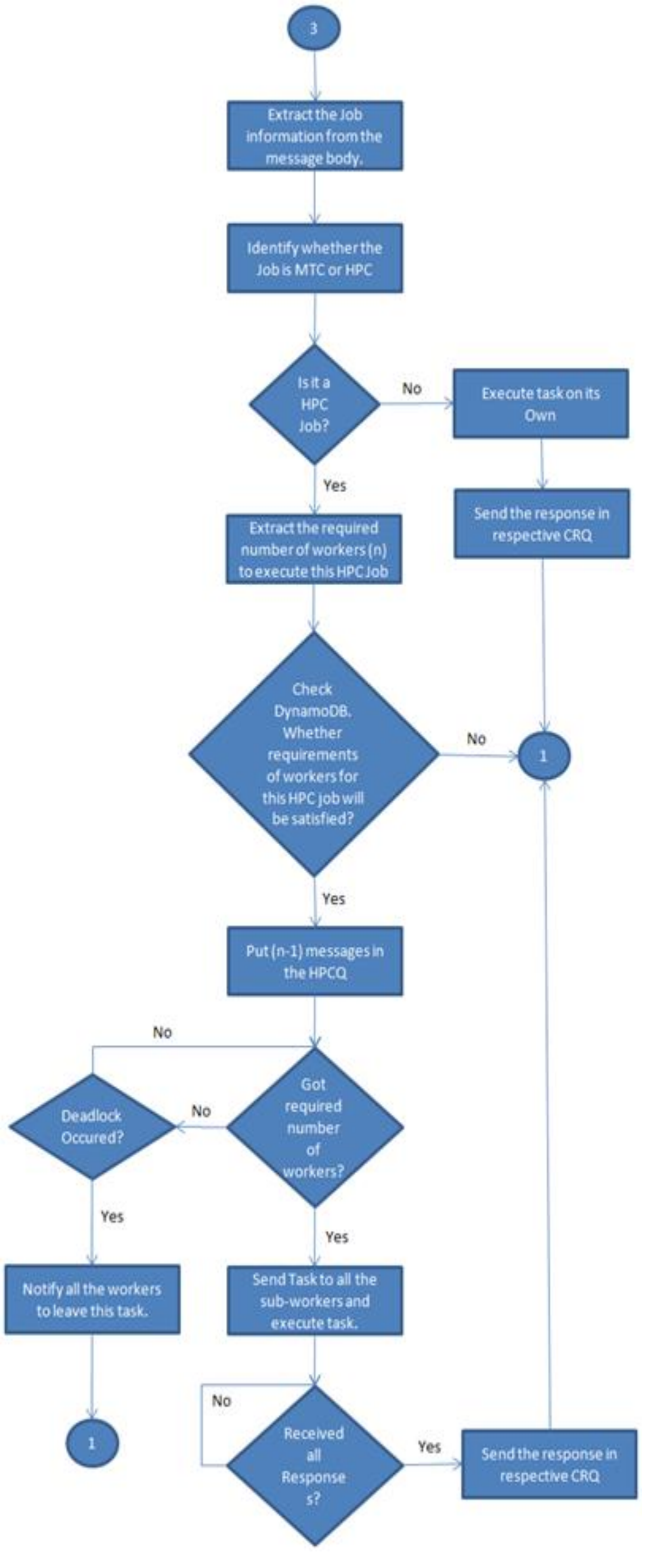
3
Extract the Job information from the message body.
Identify whether the Job is MTC or HPC
Is it a HPC Job?
No
Execute task on its Own
Send the response in respective CRQ
Yes
Extract the required number of workers (n) to execute this HPC Job
Check DynamoDB. Whether requirements of workers for this HPC job will be satisfied?
No
1
Yes
Put (n-1) messages in the HPCQ
No
Deadlock Occured?
No
Got required number of workers?
Yes
Yes
Notify all the workers to leave this task.
Send Task to all the sub-workers and execute task.
1
No
Received all Responses?
Yes
Send the response in respective CRQ

Figure 3: Main worker flow chart

Effects of Deadlock Recovery (Stale messages in the system / System Slowdown): When worker manager detects a deadlock, it will leave the current HPC task and also it will tell currently associated sub-workers to stop waiting for the task and leave it. This will lead us to a problem of stale tasks messages left in the HPCQ. Let's assume worker manager requires 'n' workers and it has got 'm' sub-workers before it detects the deadlock, then 'm' messages from HPCQ have already been deleted by respective sub-workers. Now, if worker manager detects a deadlock, then worker manager and associated sub-workers will leave this task. Due to this, though the current HPC job is aborted, 'n-m-1' messages for this HPC job, will still be there, resulting in stale messages, which slow down the process and will eventually result in deadlock. So to avoid the problem of stale messages, we have implemented two ways. First, if worker itself finds his own message in HPCQ, it will delete the message. If some other worker finds this message, it will try to connect with worker manager mentioned in the message. As this task message is no more valid and if worker is able to connect worker manager in the message, then worker will get negative response resulting in deletion of this stale message. If worker is not able to connect, then worker will increment the pick-up count. If this pick-up count reaches certain limit, this message will be deleted.

3.2.4. Sub-worker

If worker picks up a message from HPCQ, then it extracts the useful information from the message such as worker manager IP and worker manager port, which is used by sub-worker to communicate with worker manager. Also, while notifying manager that sub-worker is willing to work with him on this HPC job, it will share its IP and port number, so that worker manager can send tasks and control information to the worker. Once, worker notifies worker manager, it waits for next instructions from worker manager. Sub-worker will get two types of instructions, first where he is asked to start working on the task sent by the worker manager and after completion it sends back response accordingly to the worker manager. Second type of instruction will ask sub-worker to leave this task. If in case, something goes wrong and worker manager could not notify sub-worker, then after a timeout period, worker will ask worker manager, whether the task it is waiting for is still valid or not. If it is still valid, then, it will wait for the task else it will leave this task.

3.2.5. Worker Communication

Worker manager and sub-workers will communicate with each other using JAVA RMI[27]. Whenever any sub-worker picks up a message from HPCQ, that sub-worker will notify the respective worker manager. In response to this notification, worker manager will either send positive or negative response. If response is negative, sub-worker will straightaway leave this task. If it is positive, then sub-worker will wait for the task. Once worker manager gets all the required number of workers, including itself, it will notify each sub-worker associated with it to execute the task, which is also sent while notifying sub-worker. Once the task execution is over, sub-worker will send response back to the worker manager. If sub-worker does not get the notification for task from worker manager within the specified time limit, then sub-worker will communicate with worker manager to check whether the task, it is waiting for, is still valid or not. It will keep on waiting, if task is valid. Also, if worker manager finds that the system has entered into deadlock, it will communicate with associated sub-workers to leave the given task. If any worker faces issues while contacting some other worker due to connection issues, that worker will try two more times, even if he does not succeed, it leave that task.

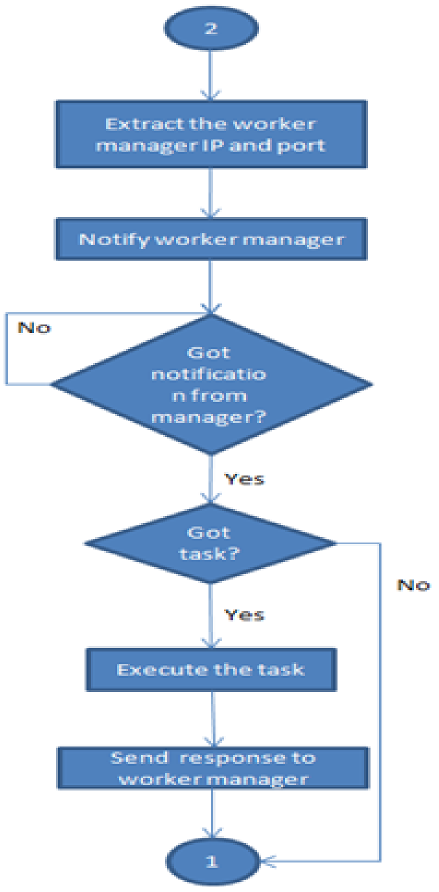
2
Extract the worker manager IP and port
Notify worker manager
No
Got notification from manager?
Yes
Got task?
No
Yes
Execute the task
Send response to worker manager
1

Figure 4: Sub-worker Flow Chart

3.2.6. Techniques Used

The project is implemented in Java. We have about 1400 lines of code including extensive comments and logs being written to files for efficient debugging. Heavy logging was needed because we sometimes faced issues with Amazon’s internal connectivity while running the tests. These logs helped us a lot in identifying where the issues occurred.

We are not implementing resource stealing so as to give every job in the system a fair chance of completion before timing out. The timeout is mainly because of connectivity issues that might occur in EC2 instances or messages in flight that are not considered by SQS while counting the number of messages SQS actually contains.

3.2.7. Major Issues

Issues we faced while using Amazon's Infrastructure:

- The major issue we faced was that we couldn't have synchronized access on DynamoDB as is implementation is abstracted from us.
- There were several issues with SQS since it will not give us accurate results on the number of messages it holds at any given point in time. This is of importance to us because we use these results to pick up a new task from the main queue or work on existing jobs that have been picked up. We are heavily dependent on these results to stop the execution of our threads.
- We have also have problem with simultaneous communication of sub workers with a worker manager. For example: Consider the scenario where the worker manager is waiting for a single sub worker, but there can be a scenario where multiple numbers of workers communicate with the worker manager. To solve this issue we had to maintain a log of number of sub workers that are communicating with the manager and this log is updated by the sub workers serially

4. EVALUATION

In this section we evaluate the performance of CloudKon-CKHPC using different metrics and also comparing it with other systems. In all of our experiments, we have used '*m1.medium*' instances for Amazon EC2, and have run all of our experiments on '*us.east.1'* datacenter of Amazon. We have used a maximum of 300 nodes and 66 SQS queues in the experiments. In order to make the experiments efficient and reliable, client and worker nodes both run on each node, i.e. within the Amazon network itself. All of the instances had Linux Operating Systems. Our framework works on any OS that has a JRE 1.6 or above running on it. We have used bash scripting language with the help of some other programs like Parallel-SSH to run the experiments. This is only to facilitate the running of all tasks in parallel on all the nodes at the same time.

The three main metrics on which we based our evaluations were: throughput, latency and number of messages overhead. We used five factors to include for each graph:

- Number of nodes in the system
- Time (in milliseconds)
- Number of workers processes
- Sleep length (in milliseconds)
- Number of tasks per job.

Throughput - The throughput is the measure of HPC jobs performed per second.

Latency - Latency is calculated to show the time interval between submitting the job and getting back a response for it.

4.1. Minimum Messages required on CloudKon-CKHPC

The figure talks about the minimum number of messages that will be generated for each HPC job. Here we are capturing the communication pattern that is a distinctive feature of our system. The number above each line in the diagram gives the number of messages that are passed and not the sequence of messages. The client submits the job to the GRQ by sending a single message to it. The worker manager will do a read and delete operation on the GRQ to get jobs. The worker manager will interact with the DynomoDB for four times in the sequence read, update, read and update for deadlock avoidance and detection. For (n-1) workers for an HPC job, the number of read and delete operations performed on the HPCQ is 2(n-1). There will be 3(n-1) messages being passed between the sub workers and the managers. The interaction between the manager and the sub worker is as follows:
Sub worker initial notification to the manager,
Task allocation from manager to sub worker,
Task completion response from sub worker to manager.
Worker managers will feed the output to the CRQ in a single message which will be picked up by the client. The client submits two messages to the CRQ for reading the values and deleting the queue.

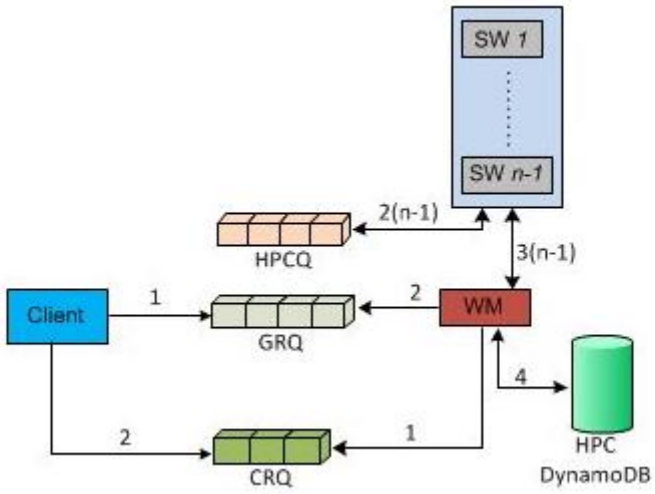

Figure 5: Message flow in the CKHPC system

Minimum number of internal messages used for the HPC implementation = **m (6n + 4)**
where,

n = Number of tasks per job
m = Number of HPC jobs

4.2. Throughput and latency on CloudKon-CKHPC

In order to measure the throughput and latency of our system we run sleep 0 tasks on worker nodes. We have evaluated the performance of CloudKon on multiple instances, starting from 1 instance and extending the experiment up to 64 nodes. We have also compared the throughput of CloudKon with SLURM and SLURM++ for up to 300 nodes. There is 1 client thread and different number of worker threads (ranging from 1 to 256) running on each instance. Each instance submits jobs ranging from 500 to 2000. On the largest scale (300 instances) our system has run 40 tasks for the comparison to other systems experiment. We have evaluated the throughput of CloudKon from 1 to 64 instances running 2 to 16 tasks. The simulations for workloads we ran are captured in the graphs below.

4.2.1. Throughput

The next two graphs shows that the throughput goes significantly higher with the increase in number of worker instances. This is because with more workers available to pick up the tasks, for the same job workload, the performance would be better. It is observed that for jobs requiring 4 tasks each, the throughput is higher as compared to jobs requiring double the tasks. Time taken to run tasks increases with the increase in the number of task requirement per job. Also if we run multiple worker processes per worker node, throughput increases with the increase number worker processes.

As observed from figure 6, with 8 tasks/job we get a very small throughput for 16 worker processes whereas for 64 processes, the throughput is around 400 jobs/sec.

Each worker node may invoke a number of worker processes to run in parallel on the same node. This graph shows that the throughput goes significantly higher with the increase in number of worker processes. This is because more workers will be available to pick up the tasks, improving performance.

There is a limit on how many number of worker processes per node can be invoked. After certain number of process per node, performance will start decreasing, which depends on the number of cores available on the system, which is the major factor for the parallelization that can be achieved on the system using multiple processes.

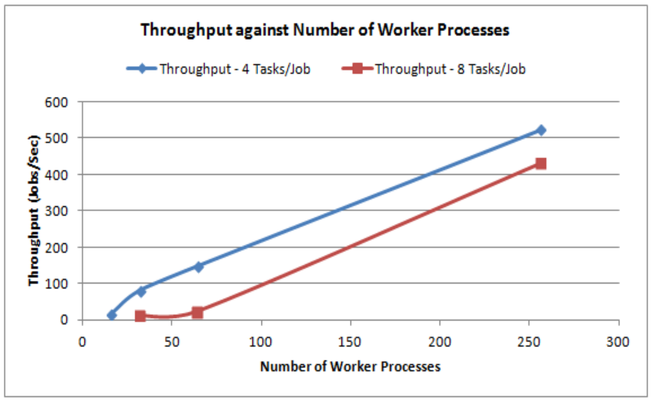

Figure 6: System throughput with different number of worker processes

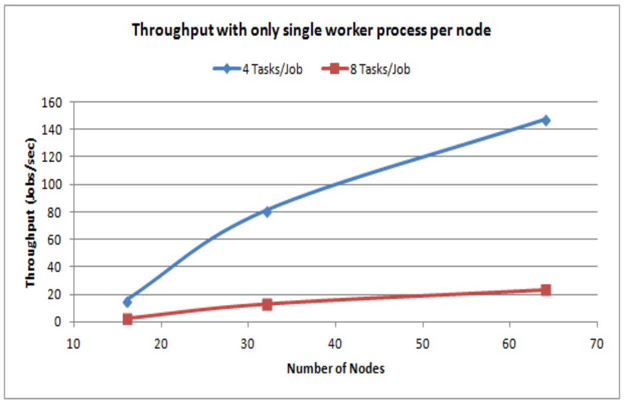

Figure 7: System throughput with single worker process per node for different nodes

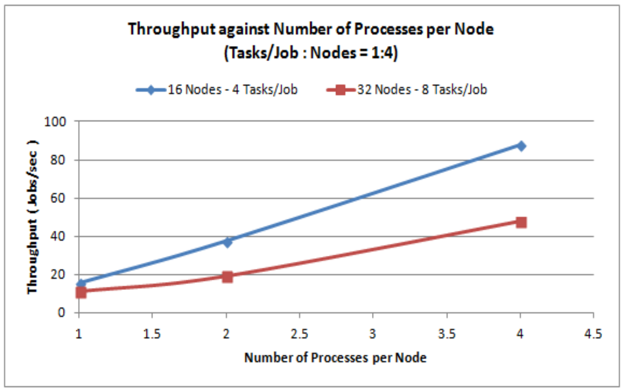

Figure 8: System throughput with different worker process per node (Tasks/Job: Nodes = 1:4)

Also, having the same number of jobs being submitted, with increase in number of tasks per job the throughput goes down as the communication cost also increases amongst the workers. Hence, time taken to run tasks increases with the increase in the number of task requirement per job.

4.2.2. Latency

With the increase in the number of nodes the latency for the system decreases as the number of jobs being submitted is same and more workers become available to carry out the same workload. Hence, we can justify the latency being higher for lower number of workers processes per node.

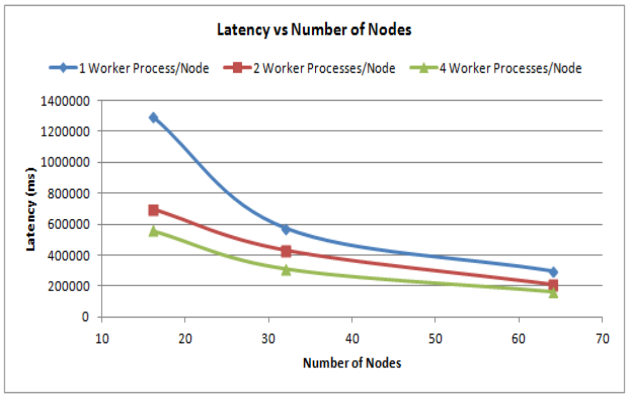

Figure 9: System throughput with single worker process per node for different nodes

As observed from Figure 9, for single worker process per node latency was 1294450 ms for 16 nodes and it went down drastically to 300864 ms for 64 nodes. Also, with the increase in number of worker processes per node the latency starts decreasing.

4.2.3. Comparison between CloudKon, CloudKon-CKHPC, Slurm, and Slurm++ for MTC tasks

To compare CloudKon-CKHPC with CloudKon we have used MTC jobs i.e.1task per job with each task being a 'Sleep 0' task. For this evaluation, 16000 MTC jobs were submitted per node. And evaluations were carried for 4 worker processes per node with number of nodes varying from 1 to 64. As seen from the figure 10, the throughput for Cloudkon is higher than the Cloudkon-CKHPC when handling MTC tasks. This is because CloudKon-CKHPC have used some additional resources to support HPC, which is the overhead to carry out MTC jobs. Throughput obtained for MTC jobs on CloudKon-CKHPC with 1 node is 157 Jobs/sec while for CloudKon it is 238 Jobs/sec. At the scale of 64 nodes, throughput for CloudKon-CKHPC is 8722 which less than throughput obtained for CloudKon which is 12947. Performance of CloudKon-CKHPC for MTC jobs is almost 2/3rd of the performance of CloudKon.

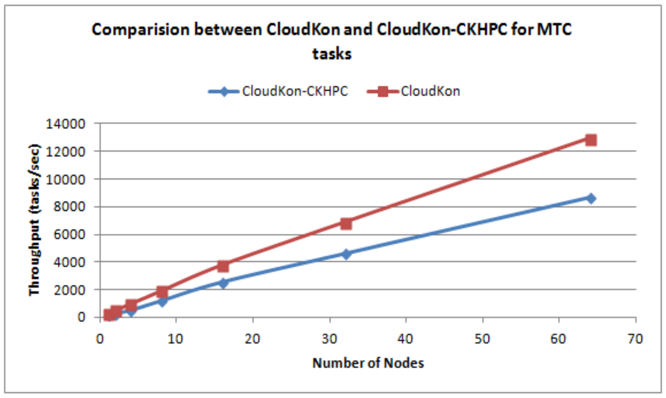

Figure 10: Comparing the Throughput for CloudKon and CKHPC

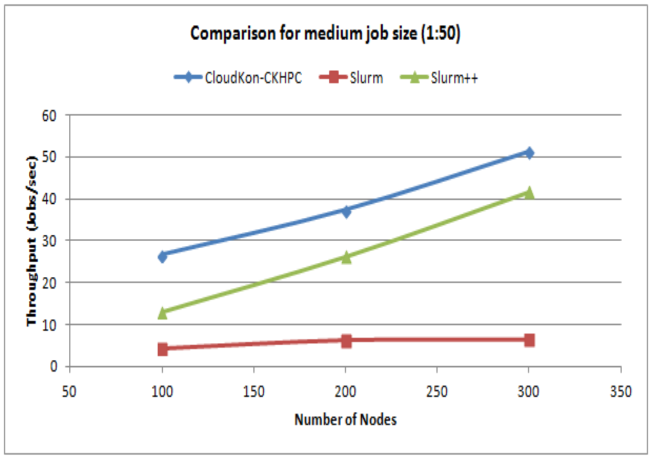

Figure 11: Comparing the Throughput for different job execution systems

We have also compared CloudKon-CKHPC with SLURM and SLRUM++. The comparison was carried at the scales of 100 nodes to 300 nodes. Also we had submitted 500 medium size jobs i.e. each job containing 1 to 50 tasks. It is observed that the throughput of CloudKon -CKHPC is higher than SLURM++ and way better than SLURM.

The results show that CloudKon-CKHPC was able to outperform the other two systems after the scale of 100 instances. SLURM performs slower than the other two systems and cannot scale up. One of the main reasons that CloudKon-CKHPC is outperforming the other two is being optimized for the public cloud environment.

5. RELATED WORK

Job management system can be centralized or hierarchical or distributed. In centralized systems, jobs are scheduled by only one centralized node on to the other nodes, where as in distributed systems each and every node maintains its own job framework. In hierarchical job scheduler, several job dispatchers are organized in the tree topology to manage jobs for their child nodes.

SLURM is an open source, centralized job management system suitable for Linux clusters of all sizes to target mainly HPC and HTC applications [2,12]. *Condor* was one of the earliest job schedulers developed to harness the unused CPU cycles on workstations for long-running batch jobs. *Falkon*, developed in 2007, also has a centralized architecture. Despite scaling and performing magnitude orders better than the state of the art, its centralized architecture hinders it from scaling to petascale systems [8].

SLURM++ [11] is a distributed job launch prototype, which extends the SLURM resource manager by integrating the ZHT [26, 28] zero-hop distributed key-value store for distributed state management. SLURM++ is comprised of multiple controllers with each one managing several SLURM daemons, while ZHT is used to store all the job metadata and the SLURM daemons' state.

All these systems lack the granularity of scheduling jobs at node/core level, which makes them incompatible for MTC applications. Moreover, they also face scalability and reliability issues as they are centralized systems.

MATRIX[10] is an execution framework that focuses on running Many Task Computing (MTC) jobs [26]. It is highly scalable and dynamic as it uses an adaptive job stealing approach. It also supports the execution of complex large-scale workflows, and has scaled up to 1K-nodes. *Sparrow* is another scheduling system that focuses on scheduling very short jobs that complete within hundreds of milliseconds [9,11]. It has a decentralized architecture making it highly scalable and also has a good load balancing strategy with near optimal performance using a randomized sampling approach[14,15,16].

Most of the above light-weight task execution frameworks have been developed from scratch, hence having tens of thousands of lines of code. Due to such large code bases, maintenance of these systems is difficult as well as expensive, and also its much harder for them to evolve once initial prototypes have been completed. CloudKon aims to make use of existing distributed and scalable building blocks to deliver an extremely compact distributed task execution framework that maintains the same levels of performance as the state-of-the-art systems.

To overcome the limitations of SLURM and other systems introduced above, CloudKon has been developed which adds support for MTC applications providing fine granularity for scheduling jobs at the node/core level. Amazon SQS used in the CloudKon guarantees the scalability and reliability as it is scalable and will deliver each message at least once. Though CloudKon supports MTC applications and overcomes limitations of SLURM, it lacks support for HPC applications and MPI applications so far.

To our knowledge, CloudKon is the only distributed task scheduler with the ability of running both MTC and HPC tasks that is designed and optimized to perform well on public cloud environment. This eliminates the need for having expensive grids or clusters for running HPC jobs. Another benefit of the cloud services is that using those services, users can implement relatively complicated systems that are able to serve in larger scales with a very short code base in a short period of time. We are able to keep up with other systems like SLURM and SLURM++ in terms of performance and these state of the art systems are written in programming languages that run faster than Java. We have to note that our system is able to compete with these systems by running on public clouds and using web services which are slower than dedicated clusters and grids running on high speed low latency networks. We attribute this performance of our system to the distributed nature of the system and the short code base.

6. CONCLUSION AND FUTURE WORK

By the end of this project we have a set of deliverables that enable us to have a running implementation of a prototype of real time HPC tasks being supported by the CloudKon framework. The following are some of important things we learned while working on this project:

- Deadlock avoidance is better than deadlock recovery. This increases the resource utilization and has little effect on the throughput of the system. With distributed storage of system state, the adverse effect on the throughput can be minimized. We have implemented an algorithm that helps the system recover from the deadlock and restore the system to a normal state since every unutilized worker has an adverse effect on the number of tasks being executed in the system.
- The use of RMI is much better than the use of socket based connections for handling simultaneous communication with multiple sub workers. This eliminates the need for opening multiple ports and having a lot of threads listening on individual ports.
- Handling race conditions and deadlocks in a real time scheduling system.
- Invoking more processes per node gives better performance than a single process per node.
- Got good hands on experience working with Amazon web services and the Google protocol buffer.
- The use of extensive shell scripting for starting and running multiple worker instances in parallel.
- Writing modular code with high cohesion.
- Having multiple classes for decreased coupling amongst the classes in the code.

We can say that our project was a success based on the below evaluations:

- On comparing system performance with other systems like SLURM and SLURM++, higher throughputs were observed for CKHPC.
- With the implementation of deadlock detection and avoidance, the system runs consistently on scales of up to 1200 processes running on 300 nodes.

We strongly believe that with most software systems there is always space for improvement and ours is an evolving prototype and with scope for improvement in the following areas:

- Extend support for MPI applications for distributed processing.
- Make it run it for real-time tasks, instead of sleep tasks.
- Limit the use of centralized DynamoDB and build a decentralized system for deadlock avoidance and efficient handling of race conditions.
- Use Dynamo instead of DynamoDB for eventual consistency resulting in high scalability.
- Dynamic calculation of time out rather than a fixed time for better resource utilization. The dynamic calculation of time out takes into account the time required to complete the previous HPC task, number of sub workers required for last HPC task and estimated time for completion of current task.

REFERENCES

[1] P. Kogge, et. al., “Exascale computing study: Technology challenges in achieving exascale systems,” DARPA-IPTO, [2008]. http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf

[2] SLURM, Lawrence Livermore National Laboratory, [online] 2013, https://computing.llnl.gov/linux/SLURM/

[3] Amazon SQS, Amazon Web Services, [online] 2013, http://aws.amazon.com/sqs/

[4] Amazon EC2, Amazon Web Services, [online] 2013, http://aws.amazon.com/ec2/

[5] Amazon DynamoDB, Amazon Web Services, [online] 2013, http://aws.amazon.com/dynamodb

[6] 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.

[7] High Performance Computing, Techopedia, [online] 2013, http://www.techopedia.com/definition/4595/high-performance-computinghpc

[8] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. "Toward Loosely Coupled Programming on Petascale Systems", IEEE/ACM Supercomputing 2008.

[9] K. Ousterhout, P. Wendell, M. Zaharia, I. Stoica. "Sparrow: Scalable scheduling for sub-second parallel jobs", Tech. Rep. UCB/EECS-2013-29, EECS Department, University of California, Berkeley, Apr 2013.

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

[11] K. Wang, X. Zhou, H. Chen, M. Lang, I. Raicu. “Next Generation Job Management Systems for Extreme Scales”, under review at ACM HPDC 2014.

[12] M. Jette, M. Grondona. "SLURM: Simple Linux Utility for Resource Management", Cluster Conference and Expo, June 23, 2003.

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

[14] A. Sinha, L.V. Kal´e. “A load balancing strategy for prioritized execution tasks", in International Parallel Processing Symposium, pages 230–237, April 1993.

[15] M.H. Willebeek-LeMair, A.P. Reeves. “Strategies for dynamic load balancing on highly parallel computers,” in IEEE Transactions on Parallel and Distributed Systems, volume 4, September 1993.

[16] G. Zhang, et. al, “Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers,” In Proceedings of the 2010 39th International Conference on Parallel Processing Workshops, ICPPW 10, pages 436-444, Washington, DC, USA, 2010.

[17] Java RMI, UMM Directory, [online] 2013, http://directory.umm.ac.id/Networking%20Manual/OReilly.Java.Rmi.pdf.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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", Journal of Physics: Conference Series 180, 012046 2009.

[23] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, M. Wilde. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation", IEEE Workshop on Scientific Workflows, 2007.

[24] Y. Zhao, I. Raicu, I. Foster, M. Hategan, V. Nefedova, M. Wilde. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments", book chapter in Grid Computing Research Progress, ISBN: 978-1-60456-404-4, Nova Publisher 2008.

[25] I. Raicu, I. Foster, A. Szalay, G. Turcu. "AstroPortal: A Science Gateway for Large-scale Astronomy Data Analysis", TeraGrid Conference 2006, June 2006

[26] 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

[27] I. Raicu, C. Dumitrescu, I. Foster. "Dynamic Resource Provisioning in Grid Environments", TeraGrid Conference 2007

[28] T. Li, R. Verma, X. Duan, H. Jin, I. Raicu. “Exploring Distributed Hash Tables in High-End Computing”, ACM Performance Evaluation Review (PER), 2011