Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing

Iman Sadooghi, Sandeep Palur, Ajay Anthony, Isha Kapur, Karthik Belagodu, Pankaj Purandare, Kiran Ramamurty, Ke Wang, Ioan Raicu
Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA
isadoogh@iit.edu, {psandeep, aanthon2, ikapur, kbelgodu, ppuranda, kramamu1, kwang22}@hawk.iit.edu, iraicu@cs.iit.edu

Abstract— **Task scheduling and execution over large scale, distributed systems plays an important role on achieving good performance and high system utilization. Due to the explosion of parallelism found in today's hardware, applications need to perform over-decomposition to deliver good performance; this over-decomposition is driving job management systems' requirements to support applications with a growing number of tasks with finer granularity. Our goal in this work is to provide a compact, light-weight, scalable, and distributed task execution framework (CloudKon) that builds upon cloud computing building blocks (Amazon EC2, SQS, and DynamoDB). Most of today's state-of-the-art job execution systems have predominantly Master/Slaves architectures, which have inherent limitations, such as scalability issues at extreme scales and single point of failures. On the other hand distributed job management systems are complex, and employ non-trivial load balancing algorithms to maintain good utilization. CloudKon is a distributed job management system that can support both HPC and MTC workloads with millions of tasks/jobs. We compare our work with other state-of-the-art job management systems including Sparrow and MATRIX. The results show that CloudKon delivers better scalability compared to other state-of-the-art systems for some metrics – all with a significantly smaller code-base (5%).**

Keywords-CloudKon, Many-Task Computing, distributed scheduling, distributed HPC scheduling

I. Introduction

The goal of a job scheduling system is to efficiently manage the distributed computing power of workstations, servers, and supercomputers in order to maximize job throughput and system utilization. With the dramatic increase of the scales of today's distributed systems, it is urgent to develop 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 [1].

Unfortunately, today's schedulers have centralized Master/Slaves architecture (e.g. Slurm [2], Condor [3][4], PBS [5], SGE [6]), where a centralized server is in charge of the resource provisioning and job execution. This architecture has worked well in grid computing scales and coarse granular workloads [7], but it has poor scalability at the extreme scales of petascale systems with fine-granular workloads [8][9]. The solution to this problem is to move to the decentralized architectures that avoid using a single component as a manager. Distributed schedulers are normally implemented in either hierarchical [10] or fully distributed architectures [31] to address the scalability issue. Using new architectures can address the potential single point of failure and improve the overall performance of the system up to a certain level, but issues can arise in distributing the tasks and load balancing among the nodes [25].

The idea of using cloud services for high performance computing has been around for several years, but it has not gained traction primarily due to many issues. Having extensive resources, public clouds could be exploited for executing tasks in extreme scales in a distributed fashion. Our goal in this project is to provide a compact and lightweight distributed task execution framework that runs on the Amazon Elastic Compute Cloud (EC2) [17], by leveraging complex distributed building blocks such as the Amazon Simple Queuing Service (SQS) [18] and the Amazon distributed NoSQL key/value store (DynamoDB) [33].

There have been many research works about utilizing public cloud environment on scientific computing and High Performance Computing (HPC). Most of these works show that cloud was not able to perform well running scientific applications [11][12][13][14]. Most of the existing research works have taken the approach of exploiting the public cloud using as a similar resource to traditional clusters and super computers. Using shared resources and virtualization technology makes public clouds totally different than the traditional HPC systems. Instead of running the same traditional applications on a different infrastructure, we are proposing to use the public cloud service based applications that are highly optimized on cloud environment. Using public clouds like Amazon as a job execution resource could be complex for end-users if it only provided raw Infrastructure as a Service (IaaS) [34]. It would be very useful if users could only login to their system and submit jobs without worrying about the resource management.

Another benefit of the cloud services is that using those services, users can implement relatively complicated systems with a very short code base in a short period of time. Our goal is to show evidence that using these services we are able to provide a system that provides high quality service that is on par with the state of the art systems in with a significantly smaller code base. ***In this paper, we design and implement a scalable task execution framework on Amazon cloud using different AWS cloud services, and***

aimed it at supporting both many-task computing and high-performance workloads.

The most important component of our system is Amazon Simple Queuing Service (SQS) which acts as a content delivery service for the tasks, allowing clients to communicate with workers efficiently, asynchronously, and in a scalable manner. Amazon DynamoDB is another cloud service that is used to make sure that the tasks are executed exactly once (this is needed as Amazon SQS does not guarantee exactly-once delivery semantics). We also leverage the Amazon Elastic Compute Cloud (EC2) to manage virtual resources. With SQS being able to deliver extremely large number of messages to large number of users simultaneously, the scheduling system can provide high throughput even in larger scales.

Today's data analytics are moving towards interactive shorter jobs with higher throughput and shorter latency [35][10]. More applications are moving towards running higher number of jobs in order to improve the application throughput and performance. A good example for this type of applications is Many-Task Computing (MTC) [15][16][39][40]. MTC applications often demand a short time to solution and may be communication intensive or data intensive [41].

As we mentioned above, running jobs in extreme scales is starting to be a challenge for current state of the art job management systems that have centralized architecture. On the other hand, the distributed job management systems have the problem of low utilization because of their poor load balancing strategies. ***We propose CloudKon as a job management system that achieves good load balancing and high system utilization at large scales.*** Instead of using techniques such as random sampling, CloudKon uses distributed queues to deliver the tasks fairly to the workers without any need for the system to choose between the nodes. The distributed queue serves as a big pool of tasks that is highly available. The worker gets to decide when to pick up a new task from the pool. This approach brings design simplicity and efficiency. Moreover, taking this approach, the system components are loosely coupled to each other. Therefore *the system will be highly scalable, robust, and easy to upgrade.* Although the motivation of this work is to support MTC tasks, it also provides support for distributed HPC scheduling. This enables CloudKon to be even more flexible running different type of workloads at the same time.

The main contributions of this work are:

1. ***Design and implement a simple light-weight task execution framework using Amazon Cloud services (EC2, SQS, and DynamoDB) that supports both MTC and HPC workloads***
2. ***Deliver good performance with <5% codebase: CloudKon is able to perform up to 1.77x better than MATRIX and Sparrow with less than 5% codebase.***
3. ***Performance evaluation up to 1024 instance scale comparing against Sparrow and MATRIX: CloudKon is able to outperform the other two systems after 64 instances scale in terms of throughput and efficiency.***

The remaining sections of this paper are as follows. Section II discusses about the design and implementation details of CloudKon. Section III evaluates the performance of the CloudKon in different aspects using different metrics. Section IV studies the related work in the area of task execution systems. Finally section V discusses about the limitations of the current work, and covers the future directions of this work.

II. Design and Implementation of CloudKon

The goal of this work is to implement a job scheduling/management system that satisfies *four major objectives*:

- ***Scale:*** *Offer increasing throughput with larger scales through distributed services*
- ***Load Balance:*** *Offer good load balancing at large scale with heterogeneous workloads*
- ***Light-weight:*** *The system should add minimal overhead even at fine granular workloads*
- ***Loosely Coupled:*** *Critical towards making the system fault tolerant and easy to maintain*

In order the achieve scalability, CloudKon uses SQS which is distributed and highly scalable. As a building block of CloudKon, SQS can upload and download large number of messages simultaneously. The independency of the workers and clients makes the framework perform well on larger scales. In order to provide other functionalities such as monitoring or task execution consistency, CloudKon also uses cloud services such as DynamoDB that are all fully distributed and highly scalable.

Using SQS as a distributed queue enables us to use pulling for load balancing and task distribution. Instead of having an administrator component (often times centralized) to decide how to distribute the jobs between the worker nodes, the worker nodes decide when to pull the jobs and run them. This would distributes the decision making role from one central node to all of the workers. Moreover, it reduces the communication overhead. In the pushing approach the decision maker has to communicate with the workers periodically to update their status and make decisions as well as distributing the jobs to among the workers. On pulling approach the only communication required is pulling the jobs. Using this approach can deliver good load balancing on worker nodes.

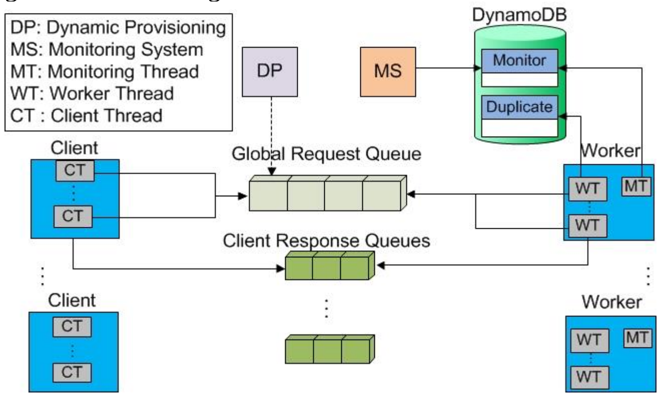

Figure 1. CloudKon architecture overview

Due to using cloud services, the CloudKon processing overhead is very low. Many of the program calls in CloudKon are the calls to the cloud services. Having totally independent workers and clients, CloudKon does not need to keep any information of its nodes such as the IP address or any other state of its nodes.

CloudKon components can operate independently with the SQS component in the middle to decouple different parts of the framework from each other. That makes our design compact, robust and easily extendable.

The scheduler can work in a cross-platform system with ability to serve on a heterogeneous environment that has systems with various types of nodes with different platforms and configurations. Using distributed queues also helps reducing the dependency between clients and the workers. The clients and workers can modify their pushing/pulling rate independently without any change to the system.

All of the advantages mentioned above rely on a distributed queue that could provide good performance in any scale. Amazon SQS is a highly scalable cloud service that can provide all of the features required to implement a scalable job scheduling system. Using this service, we can achieve the goal of having a system that perfectly fits in the public cloud environment and runs on its resources optimally.

The system makes it easy for the users to run their jobs over the cloud resources in a distributed fashion just using a client front end without the need to know about the details of the underlying resources and need to set up and configure a cluster.

A. Architecture

This section explains about the system design of CloudKon. We have used a component based design on this project for two reasons. (1) A component based design fits better in the cloud environment. It also helps designing the project in a loosely-coupled fashion. (2) It will be easier to improve the implementation in the future.

The following sections explain the system architecture for both MTC and HPC workloads. CloudKon has the ability to run workloads with a mixture of both task types. The first section shows the system architecture in case of solely running MTC tasks. The second section describes the process in case of running HPC tasks.

1) MTC task management

Figure 1 shows the different components of CloudKon that are only involved with running MTC tasks. An MTC task is defined to be a task that requires computational resources that can be satisfied by a single worker (e.g. where the worker manages either a core or a node). The client node works as a front end to the users to submit their tasks. SQS has a limit of 256 KB for the size of the messages which is sufficient for CloudKon Task lengths. In order to send tasks via SQS we need to use an efficient serialization protocol with low processing overhead. We use Google Protocol buffer for this reason. The Task saves the system log during the process while passing different components. Thus we can have a complete understanding of the different components using the detailed logs.

The main components of the CloudKon for running MTC jobs are Client, Worker, Global Request Queue and the Client Response Queues. The system also has a Dynamic Provisioner to handle the resource management. It also uses DynamoDB to provide monitoring. There is a monitoring thread running on each worker that periodically reports utilization of each worker to the DynamoDB key value store.

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. Having the Global Queue address is sufficient for a Client component to join the system. The Client program is multithreaded. So it can submit multiple tasks in parallel. Before sending any tasks, the Client creates a response queue for itself. All of the submitted tasks carry the address of the Client response queue. The Client has also the ability to use task bundling to reduce the communication overhead.

In order to improve the system performance and efficiency, we decided to put two modes. If the system is running MTC tasks, all of the workers work as normal task running workers. But in case of running HPC workloads or workloads with the combination of HPC and MTC tasks, other than the normal workers the workers could also become either worker managers that manage the HPC jobs or sub-workers that run the HPC tasks.

Similar to the Client component, the Worker component runs independently in the system. For MTC support, the worker functionality is relatively simple and straight forward. Having the Global request queue, the Workers can join and leave the system any time during the execution. The Global Request Queue acts as a big pool of Tasks. Clients can submit their Tasks to this queue and Workers can pull Tasks from it. Using this approach, the scalability of the system is only dependent on the scalability of the Global Queue and it will not put extra load on workers on larger scales. Worker code is also multithreaded and is able to receive multiple tasks in parallel. Each thread can pull up to 10 bundled tasks together. Again, this feature is enabled to reduce the large communication overhead. After receiving a task, the worker thread verifies the task duplication and then checks for the task type. In case of running MTC tasks, it will run it right away. Then it puts the results into the task and using the pre-specified address inside the task, it sends back the task to the Client respond queue. As soon as response queue receives a task, the corresponding client thread pulls the results. The process ends when the Client receives all of its task results.

2) HPC task management

Figure 2 shows the extra components to run HPC jobs. As mentioned above, in case of running combination of HPC and MTC jobs, each worker can have different roles. In case of receiving a MTC task the worker proceeds with doing the task by itself. DynamoDB is used to maintain the status of the system so that the workers can decide on the viability of executing a HPC task. In essence, in DynamoDB, we store the current number of running managers and the sub workers that are busy executing HPC tasks, which gives other workers insight about how many available resources exist.

If worker receives a HPC job, DynamoDB is checked to make sure that there are enough available nodes running in the system for the HPC task execution. If this is satisfied, the worker (now called as worker manager) puts n messages in a second SQS (HPC Task Queue). *n* is the number of workers needed by the worker manager to execute the task. If there are no enough available resources, the node is not allowed to carry on as worker manager; instead this node will check the HPC Task Queue and act as a sub worker. If there are messages in the HPC queue, the sub-worker will notify the manager using the worker managers IP address. The worker manager and sub-worker use RMI for communication. Worker Manager holds onto all of its sub-workers until it has enough to start the execution. After the execution, the worker manager sends the result to the response queue to be picked up by the client.

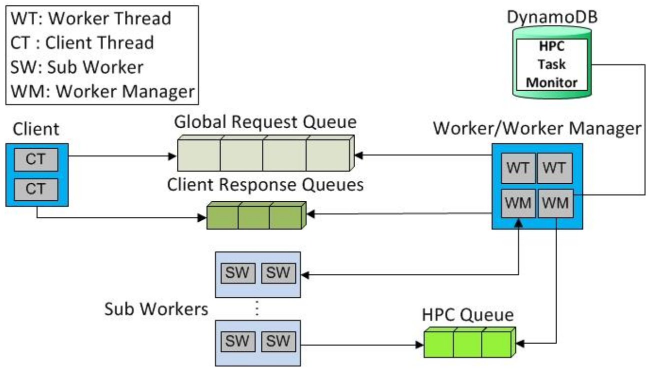

Figure 2. CloudKon-HPC architecture overview

B. *Task Execution Consistency Issues*

A major limitation of SQS is that it does not guarantee delivering the messages exactly once. It guarantees delivery of the message at least once. That means there might be duplicate messages delivered to the workers. The existence of the duplicate messages comes from the fact that these messages are copied to multiple servers in order to provide high availability and increase the ability of parallel access. We need to provide a technique to prevent running the duplicate tasks delivered by SQS. In many types of workloads running a task more than once is not acceptable. In order to be compatible for these types of applications CloudKon needs to guarantee the exactly once execution of the tasks.

In order to be able to verify the duplication we use DynamoDB. DynamoDB is a fast and scalable key-value store. After receiving a task, the worker thread verifies that if this is the first time that the task is going to run. The worker thread makes a conditional write to the DynamoDB table adding the unique identifier of the task which is a combination of the Task ID and the Client ID. The operation succeeds if the Identifier has not been written before. Otherwise the service throws an exception to the worker and the worker drops the duplicate task without running it. This operation is an atomic operation. Using this technique we have minimized the number of communications between the worker and DynamoDB.

As we mentioned above, exactly once delivery is necessary for many type of applications such as scientific applications. But there are some applications that have more relaxed consistency requirements and can still function without this requirement. Our program has ability to disable this feature for these applications to reduce the latency and increase the total performance. We will study the overhead of this feature on the total performance of the system in the evaluation section.

C. *Dynamic Provisioning*

One of the main goals in the public cloud environment is the cost-effectiveness. The affordable cost of the resources is one of the major features of the public cloud to attract users. It is very important for a Cloud-enabled system like this to keep the costs at the lowest possible rate. In order to achieve the cost-effectiveness we have implemented the dynamic provisioning [44] system. Dynamic provisioner is responsible for assigning and launching new workers to the system in order to keep up with the incoming workload.

The dynamic provisioner component is responsible for launching new worker instances in case of resource shortage. The application checks the queue length of the global request queue periodically and compares the queue length with its previous size. If the increase rate is more than the allowed threshold, it launches a new Worker. As soon as being launched, the Worker automatically joins the system. Both checking interval and the size threshold are configurable by the user.

In order to provide a solution for dynamically decreasing the system scale to keep the costs low, we have added a program to the workers that is able to terminate the instance if two conditions hold. That only happens if the worker goes to the idle state for a while and also if the instance is getting close to its lease renewal. The instances in Amazon EC2 are charged on hourly basis and will get renewed every hour of the user don't shut them down. This mechanism helps our system scale down automatically without the need to get any request from a component. Using these mechanisms, the system is able to dynamically scale up and down.

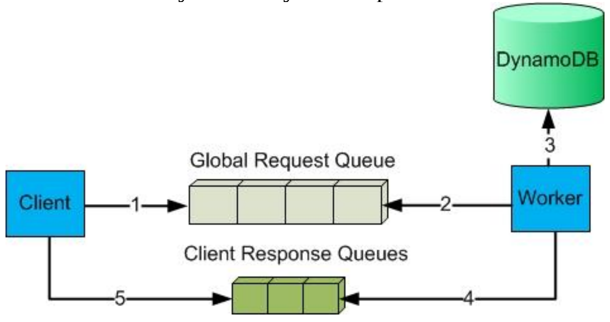

Figure 3. Communication Cost

D. *Communication Costs*

The network latency between the instances in the public Cloud is relatively high compared to HPC systems [36][37]. In order to achieve reasonable throughput and latency we need to minimize the communication overhead between the different components of the system. Figure 3 shows the number of communications required to finish a complete cycle of running a task. There are 5 steps of communication to execute a task. CloudKon also provides task bundling during the communication steps. Client can send multiple tasks together. The maximum message batch size in SQS is 256 KB or 10 messages.

E. *Security and Reliability*

For the system security of CloudKon, we rely on the security of the SQS. SQS provides a highly secure system using authentication mechanism. Only authorized users can access to the contents of the Queues. In order to keep the latency low, we don't add any encryption to the messages. SQS provides reliability by storing the messages redundantly on multiple servers and in multiple data centers [18].

F. *Implementation Details*

We have implemented all of the CloudKon components in Java. Our implementation is multithreaded in both Client and Worker component codes. Many of the features in both of these systems such as Monitoring, Consistency, number of threads and the Task bundling size is configurable as a program input argument.

Taking advantage of AWS service building blocks, our system has a short and simple code base. The code base of CloudKon is significantly shorter than other common task execution systems like Sparrow or MATRIX. CloudKon code has about 1000 lines of code (LOC), while Sparrow has 24000+ LOC, and MATRIX has 10500++ LOC. This can highlight the potential benefits of the public cloud services. We were able to create a complex and scalable system by re-using scalable building blocks in the cloud.

III. Performance Evaluation

We evaluate the performance of the CloudKon and compare it with two other distributed job management systems, namely Sparrow and MATRIX. First we discuss their high level features and major differences. Then we compare their performance in terms of throughput and efficiency. We also evaluate the latency of CloudKon.

A. *Comparing CloudKon with other Scheduling Systems*

We sufficed to compare our system with Sparrow and MATRIX as these two systems represent the best-of-breed open source distributed task management systems.

Sparrow was designed to achieve the goal of managing milliseconds jobs on a large scale distributed systesm. It uses a decentralized, randomized sampling approach to schedule jobs on worker nodes. The system has multiple schedulers that each have a list of workers and distributed the jobs among the workers deciding based on the worker's job queue length. Sparrow was tested on up to hundred nodes on the original paper.

MATRIX is a fully distributed MTC task execution fabric that applies work stealing technique to achieve distributed load balancing, and a DKVS, ZHT, to keep task metadata. In MATRIX, each computer node runs a scheduler, an executor and a ZHT server. The executor could be a separate thread in the scheduler. All the schedulers are fully-connected with each one knowing all of others. The client is a bench marking tool that issues request to generate a set of tasks, and submits the tasks to any scheduler. The executor keeps executing tasks of a scheduler. Whenever a scheduler has no more tasks to be executed, it initials the adaptive work stealing algorithm to steal tasks from candidate neighbor schedulers. ZHT is a DKVS that is used to keep the task meta-data in a distributed, scalable, and fault tolerant way.

One of the main differences between Sparrow and CloudKon or MATRIX is that Sparrow distributes the tasks by pushing them to the workers, while CloudKon and MATRIX use pulling approach. Also, in CloudKon, the system sends back the task execution results to the clients. But in both Sparrow and MATRIX, the system doesn't send any type of notifications back to the clients. That could allow Sparrow and MATRIX to perform faster, since it is avoiding one more communication step, but it also makes it harder for clients to find out if their tasks were successfully executed.

B. *Testbed*

We deploy and run all of the three systems on Amazon EC2. We have used m1.medium instances on Amazon EC2. We have run all of our experiments on the Amazon datacenter (us.east.1). We have scaled the experiments up to 1024 nodes. In order to make the experiments efficient, client and worker nodes both run on each node. All of the instances had Linux Operating Systems. Our framework should work on any OS that has a JRE 1.7, including Windows and Mac OSX.

C. *Throughput*

1) *MTC Tasks*

In order to measure the throughput of our system we run sleep 0 tasks. We have also compared the throughput of CloudKon with Sparrow and MATRIX. There are 2 client threads and 4 worker threads running on each instance. Each instance submits 16000 tasks. Figure 4 compares the throughput of CloudKon with Sparrow and MATRIX on different scales. Each instance submits 16000 tasks aggregating to 16.38 million tasks on the largest scale.

The throughput of MATRIX is significantly higher than the CloudKon and Sparrow on 1 instances scale. The reason is that MATRIX can run many fine-grained tasks locally any scheduling or network overhead. But on CloudKon the tasks must go through the network even if there is one node running on the system. The gap between the throughputs of the systems gets smaller as the network overhead adds up to the other two systems. MATRIX schedulers synchronize with each other using all-to-all synchronization method. Having too many open TCP connections by workers and schedulers on 256 instances scale leads MATRIX to be unstable. The network performance on EC2 cloud is significantly lower than that of HPC systems, where MATRIX has successfully been run at 1024-node scales.

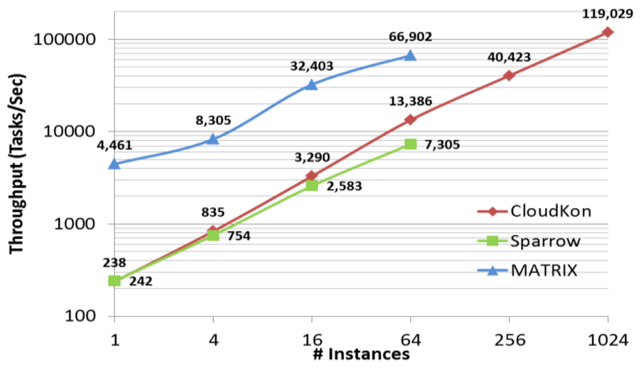

Figure 4. Throughput of CloudKon, Sparrow and MATRIX (MTC tasks)

Sparrow is the slowest among the three systems in terms of throughput. It shows a stable throughput with almost linear speedup up to 64 instances. As the number of instances increases more than 64, the list of instances to choose from for each scheduler on Sparrow increases. Therefore many workers remain idle and the throughput will not increase as expected. We were not able to run Sparrow on 128 or 256 instances scale as there were too many sockets open on schedulers resulting into system crash.

CloudKon achieves good 500X speedup starting from 238 tasks per second on 1 instance to 119K tasks per second on 1024 instances. Unlike the other two systems, the scheduling process on CloudKon is not done by the instances. Since the job management is handled by SQS, the performance of the system is mainly dependent of this service. We predict that the throughput would continue to scale until it reaches the SQS performance limits (which we were not able to reach up to 1024 instances). Due to the budget limitations, we were not able to expand our scale beyond 1024 instances, although we plan to apply for additional Amazon AWS credits and to push our evaluation to 10K instance scales, the largest allowable number of instances per user without advanced reservation.

2) *HPC Tasks*

In This section we show the throughput of the CloudKon running HPC tasks workloads. Running HPC tasks adds more overhead to the system as there will be more steps to run the tasks. Instead of running the job right away, the worker manager needs to go over a few steps and wait to get enough resources to run the job. This would slow down the system and lowers the system efficiency. But it doesn't affect the scalability. Using CloudKon can majorly improve the run time of HPC workloads by parallelizing the task execution that is normally done in a sequential fashion. We have chosen jobs with 4, 8 and 16 tasks. There are 4 worker threads running on each instance. The number of executed tasks on each scale for different workers is equal.

Figure 5 compares the system throughput in case of running HPC jobs with different number of tasks per job. The results show that the throughput of running jobs with more number of tasks per job is lower. The jobs with more tasks need to wait for more sub-workers to start the process. That adds more latency and slows down the system. We can see that CloudKon is able to achieve a high throughput of 205 jobs per second which is already much higher than what Slurm can achieve. The results also show good scalability as we add more instances.

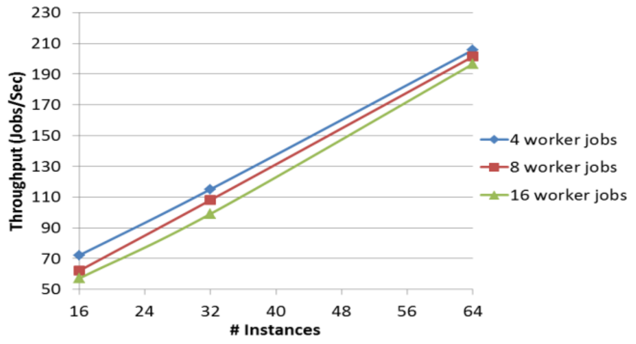

Figure 5. Throughput of CloudKon (HPC tasks)

D. *Latency*

In order to measure latency accurately, the system has to record the request and respond timestamps of each task. The problem with Sparrow and MATRIX is that on their execution process workers don't send notifications to the clients. Therefore it is not possible to measure the latency of each task comparing timestamps from different nodes. In this section we have measured the latency of CloudKon and analyzed the latency of different steps of the process.

Figure 6 shows the latency of CloudKon for sleep 0 ms scaling from 1 to 1024 instances. Each instance is running 1 client thread and 2 worker threads and sending 16000 tasks per instance.

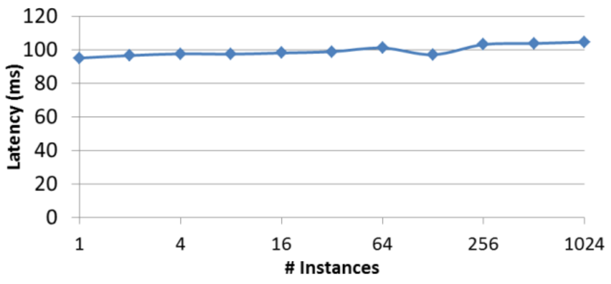

Figure 6. Latency of CloudKon sleep 0 ms tasks

The latency of the system at 1 node is relatively high showing 95 ms overhead added by the system. But this will be acceptable on larger scales. The fact that the latency doesn't increase more than 10 ms while increasing the number of instances from 1 instance to 1024 instance shows that CloudKon is stable. SQS as the task pool is a highly scalable service being backed up with multiple servers keeping the service very scalable. Thus scaling up the system by adding threads and increasing the number of tasks doesn't affect the SQS performance. The client and worker nodes always handle the same number of tasks on different scales. Therefore scaling up doesn't affect the instances. CloudKon includes multiple components and its performance and latency depends on its different components. The latency result on figure 6 does not show us any details about the system performance. In order to analyze the performance of the different components we measure the time that each task spends on different components of the system by recording the time during the execution process.

Figure 7, Figure 8, and Figure 9 respectively show the cumulative distribution of deliver-task stage, deliver-result stage, and the execute-task stage of the tasks on CloudKon. Each communication stage has three steps: sending, Queuing and receiving. The latency of the SQS API calls including send-task and receive-task on both are quite high compared to the execution time of the tasks on CloudKon. The reason for that is the expensive Web Service API call cost that uses XML format for communication. The worker takes 16ms on more than 50% of the times. This includes the DynamoDB that takes 8ms on more than 50% of the times. This shows us that hypothetically CloudKon latency can improve significantly if we use a low overhead distributed message queue that could guarantee the exactly once delivery of the tasks. We will cover this more in the future work section.

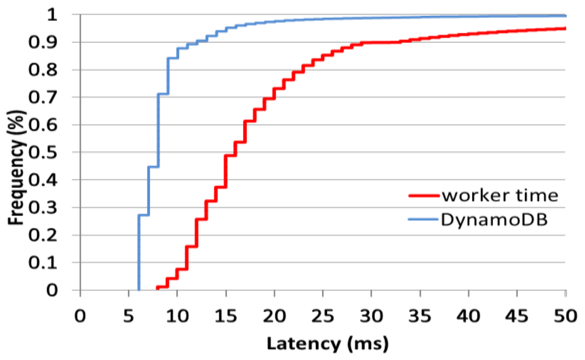

Figure 7. Cumulative Distribution of the latency on the task execution step

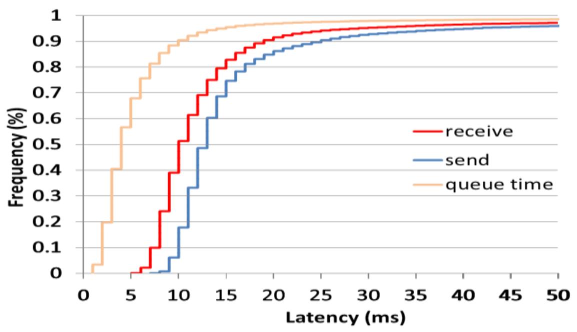

Figure 8. Cumulative Distribution of the latency on the task submit step

Another notable point is the difference between the deliver-task and deliver-result time in both Queuing and receiving back, even though they have the same API calls. The time that the tasks spend on the response-queue is longer than the time it spends on request-queue. The reason is there are two worker threads and only one client thread on each instance. Therefore the frequency of pulling tasks is higher when the tasks are pulled by the worker threads.

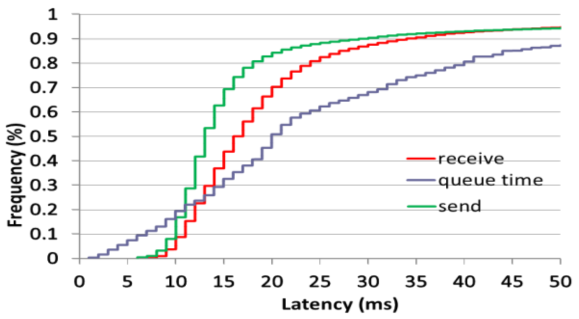

Figure 9. Cumulative Distribution of the latency on the result delivery step

E. *Efficiency of CloudKon*

It is very important for the system to manage the systems efficiently. Achieving high efficiency on distributed job scheduling systems is not trivial. It is hard to fairly distribute the workload on all of the workers and keep all of the nodes busy during the execution on larger scales.

In order to show the system efficiency we have designed two sets of experiments. We test the system efficiency in case of homogeneous and heterogeneous tasks. The homogeneous tasks have a certain task duration length. Therefore it is easier to distribute them since the scheduler assumes it takes the same time to run them. This could give us a good feedback about the efficiency of the system in case of running different task types with different granularity. We can also assess the ability of the system to run the very shot length tasks. A problem with the first experiment is that not all of the tasks take the same amount of time to run. This can hugely affect the system efficiency if the scheduler is not taking the tasks length into the consideration. Having a random workload can show how a scheduler will work in case of running real applications.

1) *Homogeneous Workloads*

In this section we compare the efficiency of CloudKon with Sparrow and MATRIX on sub second tasks. Figure 10 shows the efficiency of 1, 16 and 128ms tasks on the systems. The efficiency of CloudKon is on 1ms tasks is lower than then other two systems. As we mentioned before, the latency of CloudKon is large for very short tasks because of the significant network latency overhead added on the execution cycle. Matrix has a better efficiency on smaller scales but as the trend shows, the efficiency drops tremendously until the system crashes because of too many TCP connections on scales of 128 instances or more. On sleep 16ms tasks, the efficiency of CloudKon is around 40% which is low (compared to the other systems). The efficiency of MATRIX starts with more than 93% on one instance but again it drops to a lower efficiency than the CloudKon on larger number of instances. We can notice that the efficiency of CloudKon is very stable compared to the other two systems on different scales. That shows that CloudKon achieves a better scalability. On sleep 128ms tasks, the efficiency of CloudKon is as high as 88%. Again, the results show that the efficiency of MATRIX drops on larger scales.

Sparrow shows very good and stable efficiency running homogenous tasks up to 64 instances. The efficiency drops after this scale for shorter tasks. Having too many workers for task distribution, the scheduler cannot have a perfect load balance and some workers remain idle. Therefore the system will be under-utilized and the efficiency drops. The system crashes on scales of 128 scales or larger because of maintaining too many sockets in schedulers.

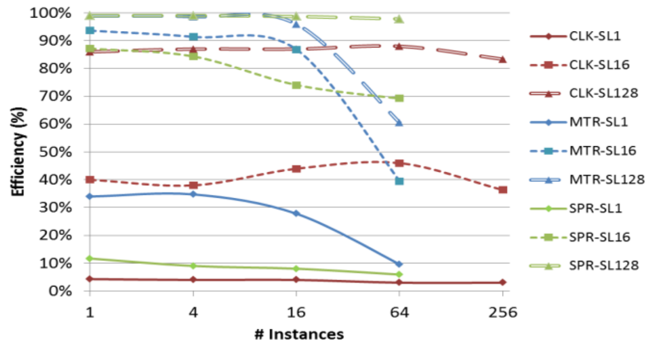

Figure 10. Efficiency of CloudKon, Sparrow and MATRIX running homogenous workloads of different task lengths (1, 16, 128ms tasks)

2) *Heterogeneous Workloads*

In order to measure efficiency, we investigated the largest available trace of real MTC workloads [38], and filtered out the logs to isolate only the sub-second tasks, which netted about 2.07M tasks with the runtime range of 1 milliseconds to 1 seconds. The tasks were submitted in a random fashion. The average task lengths of different instances are different from each other.

Each instance runs 2K tasks on average. The efficiency comparison on Figure 9 shows similar trends for CloudKon and MATRIX. On both systems the worker pulls a task only when it has available resources to run the task. Therefore the fact that the execution duration of the tasks is different does not affect the efficiency of the system. On the other hand on Sparrow, the scheduler distributes the tasks by pushing them to the workers that have less number of tasks to be executed in their queue. The fact that the tasks have different run time is going to affect the system efficiency. Some of the workers may have multiple long tasks and many other workers may have short tasks to run. Thus there will be a big imbalance among the workers with some of the being loaded with big tasks and the rest being under-utilized and the system run time will be bound to the run time of the workers with longer jobs to run.

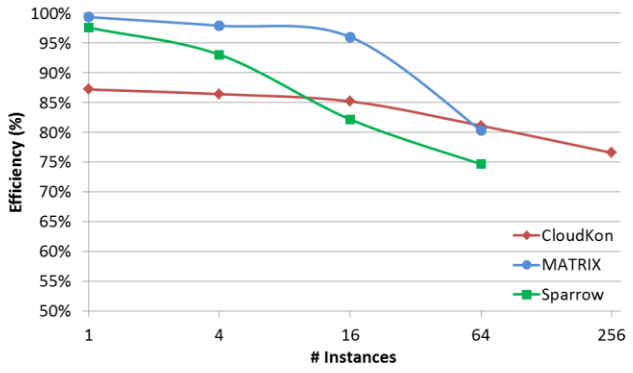

Figure 11. Efficiency of the systems running heterogeneous workloads.

Being under-utilized, the efficiency of Sparrow has the largest drop from 1 instance to 64 instances. The system was not functional on 128 instances or more. Similarly, the efficiency of MATRIX started with a high efficiency, but started to drop significantly because of too many open sockets on TCP connections. The efficiency of CloudKon is not as high as the other two systems, but it is more stable as it only drops 6% from 1 to 64 instances compared to MATRIX that drops 19% and Sparrow that drops 23%. Again, CloudKon was the only functional system on 256 instances with 77% efficiency.

F. *The overhead of consistency*

In this section we evaluate effect of tasks execution consistency on CloudKon. Figure 10 shows the system run-time for sleep 16ms with the duplication controller enabled and disabled. The overhead for other sleep tasks were similar to this experiment. So we have only included one of the experiments in this paper.

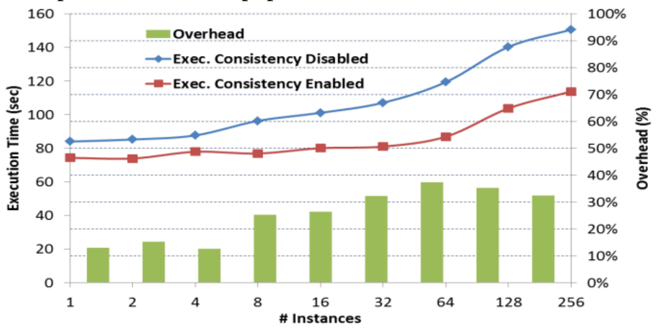

Figure 12. The overhead of task execution consistency on CloudKon

The consistency overhead increases with the scale. The inconsistency on different scales is the result of the variable number of duplicate messages on each run. That results in more random system performance on different experiments. In general the overhead on scale of less than 10 is less than 15%. This overhead is mostly for the successful write operations on DynamoDB. The probability of getting duplicate tasks increases on larger scales. Therefore there will be more exceptions. That leads to a higher overhead. The overhead on larger scales goes up to 35%. However, the overhead rate is stable and does not pass this rate. Using a distributed message queue that guarantees exactly-once delivery can improve the performance significantly.

IV. Related Work

The job schedulers could be centralized, where a single dispatcher manages the job submission, and 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.

Condor [3] was implemented to harness the unused CPU cycles on workstations for long-running batch jobs. Slurm [2] 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. Portable Batch System (PBS) [5] was originally developed to address the needs of HPC. It can manage batch and interactive jobs, and add the ability to signal, rerun and alter jobs. LSF Batch [19] is the load-sharing and batch-queuing component of a set of workload management tools.

All these systems target as the HPC or HTC applications, and lack the granularity of scheduling jobs at finer levels making them hard to be applied to the MTC applications. What's more, the centralized dispatcher in these systems suffers scalability and reliability issues. In 2007, a light-weight task execution framework, called Falkon [9] 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 [8]. A hierarchical implementation of Falkon was shown to scale to a petascale system in [8], the approach taken by Falkon suffered from poor load balancing under failures or unpredictable task execution times. Although distributed load balancing at extreme scales is likely a more scalable and resilient solution, there are many challenges that must be addressed (e.g. utilization, partitioning). Fully distributed strategies have been proposed, including neighborhood averaging scheme (ACWN) [20][21][22][23]. In [23], several distributed and hierarchical load balancing strategies are studied, such as Sender/Receiver Initiated Diffusion (SID/RID), Gradient Model and a Hierarchical Balancing Method. Other hierarchical strategies are explored in [22]. Charm++ [24] supports centralized, hierarchical and distributed load balancing. In [24], the authors present an automatic dynamic hierarchical load balancing method for Charm++, which scales up to 16K-cores on a Sun Constellation supercomputer for a synthetic benchmark.

Sparrow is another scheduling system that focuses on scheduling very short jobs that complete within hundreds of milliseconds [25]. It has a decentralized architecture that makes it highly scalable. It also claims to have a good load balancing strategy with near optimal performance using a randomized sampling approach. It has been used as a building block of other systems.

Omega presents a scheduling solution for scalable cluster using parallelism, shared-state and lock-free optimistic concurrency control [26]. The difference of this work with ours is that it optimized for course-grained scheduling of dedicated resources. CloudKon uses elastic resources. It is optimized for scheduling of both HPC and MTC tasks.

Work stealing is another approach that has been used at small scales successfully in parallel languages such as Cilk [24], to load balance threads on shared memory parallel machines [28][29][30]. However, the scalability of work stealing has not been well explored on modern large-scale systems. In particular, concerns exist that the randomized nature of work stealing can lead to long idle times and poor scalability on large-scale clusters [30]. The largest studies to date of work stealing have been at thousands of cores scales, showing good to excellent efficiency depending on the workloads [30]. MATRIX is an execution fabric that focuses on running Many Task Computing (MTC) jobs [31]. It uses an adaptive work stealing approach that makes it highly scalable and dynamic. It also supports the execution of complex large-scale workflows. Most of these existing light-weight task execution frameworks have been developed from scratch, resulting in code-bases of tens of thousands of lines of code. This leads to systems which are hard and expensive to maintain, and potentially much harder to evolve once initial prototypes have been completed. This work aims to leverage existing distributed and scalable building blocks to deliver an extremely compact distributed task execution framework while maintaining the same level of performance as the best of breed systems.

To our knowledge CloudKon is the only job management system to support both distributed MTC and HPC scheduling. We have been prototyping distributed job launch in the Slurm job resource manager under a system called Slurm++ [32], but that work is not mature enough yet to be included in this study. Moreover, CloudKon is the only distributed task scheduler that is designed and optimized to run on public cloud environment. Finally, CloudKon has an extremely compact code base, at 5% of the code base of the other state-of-the-art systems.

V. Conclusion and Future Work

Large scale distributed systems require efficient job scheduling system to achieve high throughput and system utilization. It is important for the scheduling system to provide high throughput and low latency on the larger scales and add minimal overhead to the workflow. CloudKon is a Cloud enabled distributed task execution framework that runs on Amazon AWS cloud. It is a unique system in terms of running both HPC and MTC workloads on public cloud environment. Using SQS service gives CloudKon the benefit of scalability. The evaluation of the CloudKon proves that it is highly scalable and achieves a stable performance over different scales. We have tested our system up to 1024 instances. CloudKon was able to outperform other systems like Sparrow and MATRIX on scales of 128 instances or more in terms of throughput. CloudKon achieves up to 87% efficiency running homogeneous and heterogeneous fine granular sub-second tasks. Compared to the other systems like Sparrow, it provides lower efficiency on smaller scales. But on larger scales, it achieves a significantly higher efficiency.

There are many directions for the future work. One direction is to make the system fully independent and test it on different public and private clouds. We are going to implement a SQS like service with high throughput at the larger access scales. With help from other systems such as ZHT distributed hash table [32], we will be able implement such a service. Another future direction of this work is to implement a more tightly coupled version of CloudKon and test it on supercomputers and HPC environments while running HPC jobs in a distributed fashion, and to compare it directly with Slurm and Slurm++ in the same environment. We also plan to explore porting some real programming frameworks, such as the Swift parallel programming system

[40][42][43] or the Hadoop MapReduce framework, which could both benefit from a distributed scheduling run-time system. This work could also expand to run on heterogeneous environments including different public and private clouds. In that case, the system can choose among different resources based on the resource cost and performance and provide optimized performance with the minimum cost.

ACKNOWLEDGEMENTS

This work was made possible in part due to the Amazon AWS Research Grants.

REFERENCES

[1] P. Kogge, et. al., “Exascale computing study: Technology challenges in achieving exascale systems,” 2008.

[2] M. A. Jette et. al, “Slurm: Simple linux utility for resource management”. In *Lecture Notes in Computer Sicence: Proceedings of Job Scheduling Strategies for Praral1el Procesing* (JSSPP) 2003 (2002), Springer-Verlag, pp. 44-60.

[3] 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.

[4] J. Frey, T. Tannenbaum, I. Foster, M. Frey, S. Tuecke. “Condor-G: A Computation Management Agent for Multi-Institutional Grids,” *Cluster Computing*, 2002.

[5] B. Bode et. al. “The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters,” *Usenix, 4th Annual Linux Showcase & Conference*, 2000.

[6] W. Gentzsch, et. al. “Sun Grid Engine: Towards Creating a Compute Power Grid,” *1st International Symposium on Cluster Computing and the Grid* (CCGRID’01), 2001.

[7] C. Dumitrescu, I. Raicu, I. Foster. “Experiences in Running Workloads over Grid3”, *The 4th International Conference on Grid and Cooperative Computing* (GCC 2005), 2005

[8] I. Raicu, et. al. “Toward Loosely Coupled Programming on Petascale Systems,” *IEEE/ACM Super Computing Conference* (SC’08), 2008.

[9] I. Raicu, et. al. “Falkon: A Fast and Light-weight tasK executiON Framework,” *IEEE/ACM SC 2007*.

[10] S. Melnik, A. Gubarev, J. J. Long, G. Romer,S. Shivakumar, M. Tolton, and T. Vassilakis. “Dremel: Interactive Analysis of Web-Scale Datasets. Proc.” *VLDB Endow*., 2010

[11] L. Ramakrishnan, et. al. “Evaluating Interconnect and virtualization performance for high performance computing”, *ACM Performance Evaluation Review*, 40(2), 2012.

[12] P. Mehrotra, et. al. “Performance evaluation of Amazon EC2 for NASA HPC applications”. In *Proceedings of the 3rd workshop on Scientific Cloud Computing* (ScienceCloud '12). ACM, NY, USA, pp. 41-50, 2012.

[13] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. “Case study for running HPC applications in public clouds,” In *Proc. of ACM Symposium on High Performance Distributed Computing*, 2010.

[14] G. Wang and T. S. Eugene. “The Impact of Virtualization on Network Performance of Amazon EC2 Data Center”. *In IEEE INFOCOM*, 2010.

[15] I. Raicu, Y. Zhao, I. Foster, “Many-Task Computing for Grids and Supercomputers,” *1st IEEE Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS) 2008.

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

[17] Amazon Elastic Compute Cloud (Amazon EC2), Amazon Web Services, [online] 2013, http://aws.amazon.com/ec2/

[18] Amazon SQS, [online] 2013, http://aws.amazon.com/sqs/

[19] LSF: http://platform.com/Products/TheLSFSuite/Batch, 2012.

[20] L. V. Kal´e et. al. “Comparing the performance of two dynamic load distribution methods,” *In Proceedings of the 1988 International Conference on Parallel Processing*, pages 8–11, August 1988.

[21] W. W. Shu and L. V. Kal´e, “A dynamic load balancing strategy for the Chare Kernel system,” *In Proceedings of Supercomputing* ’89, pages 389–398, November 1989.

[22] A. Sinha and L.V. Kal´e, “A load balancing strategy for prioritized execution of tasks,” In International Parallel Processing Symposium, pages 230–237, April 1993.

[23] 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

[24] 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.

[25] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. “Sparrow: distributed, low latency scheduling”. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (SOSP '13). ACM, New York, NY, USA, 69-84.

[26] M. Schwarzkopf, A Konwinski, M. Abd-el-malek, and J. Wilkes, Omega: Flexible, scalable schedulers for large compute clusters. *In Proc. EuroSys* (2013).

[27] Frigo, et. al, “The implementation of the Cilk-5 multithreaded language,” *In Proc. Conf. on Prog. Language Design and Implementation* (PLDI), pages 212–223. ACM SIGPLAN, 1998.

[28] R. D. Blumofe, et. al. “Scheduling multithreaded computations by work stealing,” *In Proc. 35th FOCS*, pages 356–368, Nov. 1994.

[29] V. Kumar, et. al. “Scalable load balancing techniques for parallel computers,” *J. Parallel Distrib. Comput.*, 22(1):60–79, 1994.

[30] J. Dinan et. al. “Scalable work stealing,” *In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

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

[32] T. Li, et al., “ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” *in IEEE International Parallel & Distributed Processing Symposium* (IPDPS ’13), 2013.

[33] Amazon DynamoDB (beta), Amazon Web Services, [online] 2013, http://aws.amazon.com/dynamodb

[34] P. Mell and T. Grance. “NIST definition of cloud computing.” National Institute of Standards and Technology. October 7, 2009.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” *in Proceedings of the 2nd USENIX Conference on Hot topics in Cloud Computing*, Boston, MA, June 2010.

[36] P. Mehrotra, et al. 2012. “Performance evaluation of Amazon EC2 for NASA HPC applications” In (ScienceCloud '12). ACM, New York, NY, pp. 41-50.

[37] I. Sadooghi, et al. “Understanding the cost of cloud computing”. Illinois Institute of Technology, Technical report. 2013

[38] I. Raicu, et al. “The Quest for Scalable Support of Data Intensive Workloads in Distributed Systems,” *ACM HPDC* 2009

[39] I. Raicu, et al. "Middleware Support for Many-Task Computing", Cluster Computing, The Journal of Networks, Software Tools and Applications, 2010

[40] Y. Zhao, et al. "Realizing Fast, Scalable and Reliable Scientific Computations in Grid Environments", book chapter in Grid Computing Research Progress, Nova Publisher 2008.

[41] I. Raicu, et al. "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

[42] Y. Zhao, et al. "Opportunities and Challenges in Running Scientific Workflows on the Cloud", IEEE CyberC 2011

[43] M. Wilde, et al. "Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers", SciDAC 2009

[44] I. Raicu, et al. "Dynamic Resource Provisioning in Grid Environments", TeraGrid Conference 2007