CloudKon Reloaded with Efficient Monitoring, Bundled Responses, and Dynamic Provisioning

Ajay Anthony[*], Sandeep Palur[*], ImanSadooghi[*], IoanRaicu[*†]
[*]Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA
[†]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL, USA
aanthon2@hawk.iit.edu, psandeep@hawk.iit.edu, isadoogh@hawk.iit.edu, iraicu@cs.iit.edu

Abstract *- In today's world the emphasis is on distributed systems which plays an important role on achieving good performance , high system utilization and scalability. 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. 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 distributed HPC and MTC scheduling, running millions of tasks on multiple nodes. CloudKon, a compact, light-weight, scalable, and distributed task execution framework (CloudKon) that builds upon cloud computing building blocks (Amazon EC2, SQS, and DynamoDB) has been developed to support high performance , high system utilization and scalability, however with some challenges and drawbacks. The downsides lie in Worker-Client communication, Monitoring system causing communication overhead and resource contention respectively. This may prove to be a potential bottleneck at higher scalable systems. The goal in this project is to reload or extend existing CloudKon with features like 1. Improved Concurrency 2. Bundled Response 3. Efficient Monitoring to address the existing challenges in CloudKon as well as implementation of dynamic provisioning.

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 [26].

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) [18], by leveraging complex distributed building blocks such as the Amazon Simple Queuing Service (SQS) [19] and the Amazon distributed NoSQL key/value store (DynamoDB) [34].

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) [35]. 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 that are able to serve in larger scales 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. To our knowledge, CloudKon [15] is the only distributed task scheduler with the ability of running both MTC [16] and HPC tasks that is designed and optimized to perform well on public cloud environment.

In this paper, we design and implement a scalable task execution framework on Amazon cloud using different AWS cloud services. The most important component of our system is Amazon Simple Queuing Service (SQS) which acts as a content delivery service for the tasks. Amazon DynamoDB is another cloud service that is used to make sure that the tasks are executed exactly once. 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 a high throughput even in larger scales.

Today's data analytics are moving towards interactive shorter jobs with higher throughput and shorter latency [36][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) [16]. MTC applications often demand a short time to solution and may be communication intensive or data intensive [17]. Tasks may be small or large, uniprocessor or multiprocessor, compute-intensive or data-intensive.

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-Reloaded , which is an extension to existing CloudKon [15] and built upon the prior work of CloudKon [15], as a job management system that achieves good load balancing and high system utilization on large scales with extended features viz. 1. Improved Concurrency 2. Bundled Response 3. Efficient Monitoring , which address the existing challenges in CloudKon [15]. Instead of using trivial techniques such as random sampling or hierarchical system design, CloudKon [15] 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.

The main contributions of this work are:

1. *Extending the existing CloudKon framework with improved level of concurrency at client and server side for homogenous tasks.*
2. *Appending response bundling of tasks at server side to client response queue reducing communication overhead as an addition to existing CloudKon framework.*
3. *Reloading the existing CloudKon framework with an efficient monitoring feature that reduces the resource contention and communication overhead.*
4. *Performance evaluation from 1 thru 1024 instances scale on Cloudkon reloaded framework.*
5. *Contribution to evaluation of throughput and efficiency to CloudKon paper submitted to CCGRID 2014.*

II. Proposed Solution

We have designed the reloaded CloudKon framework with the following improvements:

1. Improved Concurrency

2. Response bundling

3. Efficient Monitoring

Having these improvements has major focus we have designed a new architecture. This section explains about the system design of reloaded 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.

Figure 1 shows the different components of CloudKon .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 and a Monitoring System to monitor the system utilization.

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 Request Queue address is sufficient for a client component to join the system. The client program is multithreaded. The number of threads can be configured by

the user or also be made dynamic depending on the number of cores in the system on which the client is running. So it can submit tasks in parallel. Before sending any tasks, the client creates a response queue for itself. All of the submitted tasks carry the address of its Client Response Queue. The client has also the ability to use task bundling to reduce the communication overhead. The client has one more level threading, created by client workers which is used only when it pulls the results back from its response queue. While pulling back results from the response queue the client workers pulls a message bundle and creates task threads that does the actual work. So that the client worker need not wait until all the messages in the bundle are deleted from the queue and stored in the list. The task thread takes this responsibility. Task threads contribute a lot to the increase in the throughput and decrease in latency and high concurrency. The task threads run in Maximum Concurrency Mode.

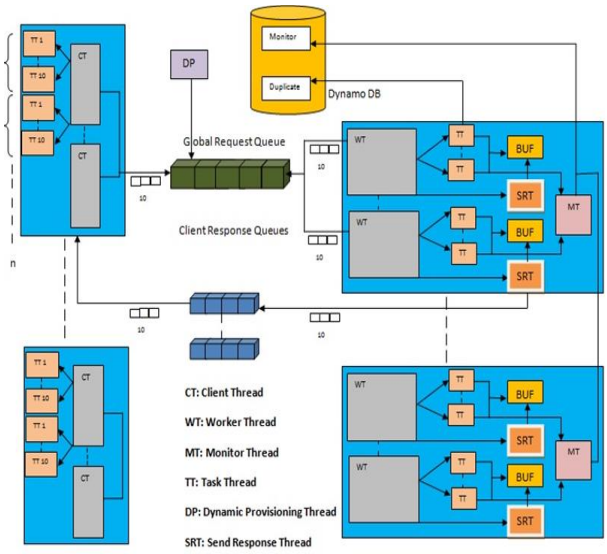

Figure 1.CloudKon Reloaded Architecture

Similar to the Client component, the Worker component runs independently in the system. 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 Request 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. The number of threads can be configured by the user or also be made dynamic, depending on the number of cores in the system on which the client is running. So it can submit 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 pulling a bundle of tasks from Global Request Queue the worker thread creates task threads that does the actual work.It deletes the task from the global queue and checks with the DyanamoDB for duplication and then execute the actual task and writes the result to the client specific array in the buffer. The task threads run in Optimal Concurrency Mode. The Send Response Thread sleeps and periodically empties the buffer and sends all the results to the corresponding Client Response Queue in bundles. Thus reduces the network overhead and also utilizes the network bandwidth efficiently as results (maximum of 10 at a time) to the same client are bundled together and sent at one time. After which the Client will be able to pull the results from its response queue.

A. Concurrency

We increase the level of concurrency on both the server and client by adding one more level of threading called Task Threads (TT).We also have two modes in which the task threads can be made to run: Optimal Concurrency Mode and Maximum Concurrency Mode. If the threads don't do I/O, synchronization, etc., and does only computation, 1 thread per core will get you the best performance. On the Client side there is going to be no computation so we run task threads on the Client side in Maximum Concurrency Mode, but on the Server side there is going to be both computation and also uses system services. So we run task threads on the Server side in Optimal Concurrency Mode

1. Optimal Concurrency Mode 1: Figure 2 shows the architecture of Optimal Concurrency Mode. In this mode we control the number of task threads running concurrently. That is we set a limit. The number can be configured or also be made dynamic .The optimal number of threads is not same for all tasksand is not always proportional to the number of cores in the system. We use this mode on server side for sleep 0 tasks for our benchmarking.

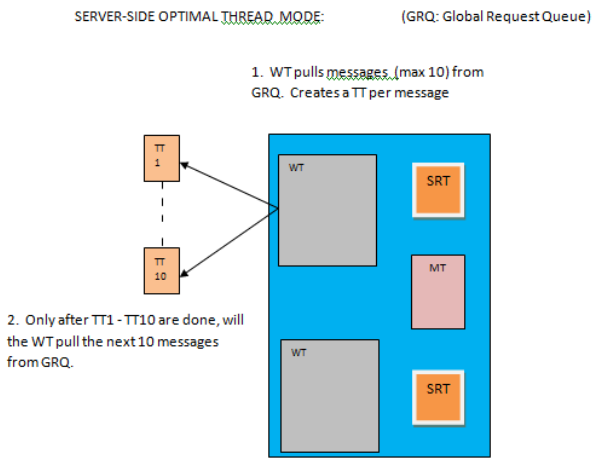

Figure 2 Optimal Concurrency Model

2. Maximum Concurrency Mode 2: Figure 3 shows the architecture of Optimal Concurrency Mode. In this mode we don't control the number of threads running parallel. We just keep on creating threads .This mode can be used only with tasks that have more I/O operations and dependencies on system services. We use this mode on client side for sleep 0 tasks for our benchmarking. As there

is no computation on the client side, we achieve very high utilization.

In this section we evaluate the performance of the CloudKon. We evaluate the performance on different metrics such as throughput, efficiency, consistency, utilization, latency. We compare CloudKon performance with two other distributed job management systems as well.

Table 1 shows one of the experiments we ran to compare the Maximum Concurrency Mode and Optimal Concurrency Mode.

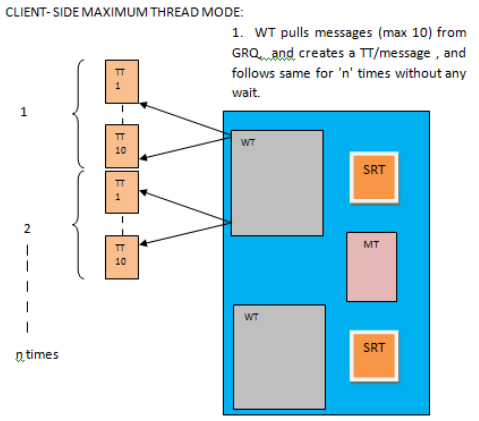

Figure 3 Maximum Concurrency Mode

Table 1 Experiments comparing the Maximum Concurrency Mode and Optimal Concurrency Mode.

Test 1: Simple Addition task of 2 numbers + sleep 100 task.

No. of Messages	**MODE 1** (seconds)	**MODE 2** (seconds)
10,000	56.963	24.254
50,000	288.548	121.202

Test 2: Simple Addition task of 2 numbers

No. of Messages	**MODE 1** (seconds)	**MODE 2** (seconds)
10,000	29.429	25.652
50,000	126.340	125.073

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 system. Dynamic Provision is responsible for assigning and launching new workers to the system in order to keep up with the incoming workload.

We first considered using Amazon Cloud Watch for this purpose. Amazon CloudWatch provides monitoring for AWS cloud resources and the applications customers run on AWS. Users can use it to collect and track metrics. The problem with using Cloud Watch in our system is that the shortest period for updating the state of the SQS is 5 minutes which makes the implementation slow to respond to changes in workloads. This is not acceptable for our application requirements running MTC and HPC tasks.

We decided to implement our own dynamic provision which takes care of 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 use 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.

D.Monitoring

Monitoring is useful for many purposes such as utilization monitoring and debugging in job management systems. CloudKon uses DynamoDB to provide monitoring. There is a monitoring thread running on each worker that periodically reports utilization of each worker to the key value store. The key value store in DynamoDB keeps track of all of the workers. The monitoring component reads the specific data it needs from the store in a real time fashion. Here we have only one monitoring thread per instance irrespective of number of worker threads running on the instance .So that we could reduce a lot of contention as all the monitor threads writes an update to the DynamoDB every second.

E.Implementation Details

We have implemented all of the CloudKon components in Java. Our implementation is multithreaded and has two levels of threading 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 Falkon, Sparrow or MATRIX. CloudKon code has about 1000 lines of code, while Falkon has 33000+ lines, Sparrow has 24000+ lines of code, and MATRIX has 10500++ lines of code. This can highlight the potential benefits of the public cloud services. We were able to create a fairly complicated and scalable system by re-using scalable building blocks in the cloud.

III. Evaluation

A. Testbed

We deployed and ran CloudKon on Amazon EC2 instances. We have used m1.large instances on Amazon EC2. We have run all of our experiments on us.east.1 datacenter of Amazon. We have scaled the experiments up to 1024 nodes. In order to make the experiments efficient, client and worker nodes both run on same node. All of the instances had Linux Operating Systems. Our framework works on any OS that has a JRE 1.7. We have used Bash scripting language for calculating throughput, latency, file transfer from EC2 instances, Parallel-SSH for parallel execution of client and server code on EC2 instances , EC2 CLI (Command Line Interface) for EC2 instance startup, termination, get EC2 IP address, etc, and AWS CLI (Command Line Interface) for SQS operations and EC2 dynamic instance startup in Dynamic Provisioning.

B. Throughput

In order to measure the throughput of our system we run sleep 0 tasks. There are 2 client threads and 4 worker threads running on each instance. Each instance submits 16000 tasks.

Figure 4provides the throughput of CloudKon on different scales. Each instance submits 16000 tasks aggregating to 16.38 million tasks on the largest scale. CloudKon achieves almost linear throughput starting from 238 tasks per second on 1 instance to 119K tasks per second on 1024 instances. CloudKon is not done by these 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 continue to scale until it reaches the SQS performance limits. Due to the budget limitation and AWS policies for normal users, we were not able to expand our scale more than 1024 instances.

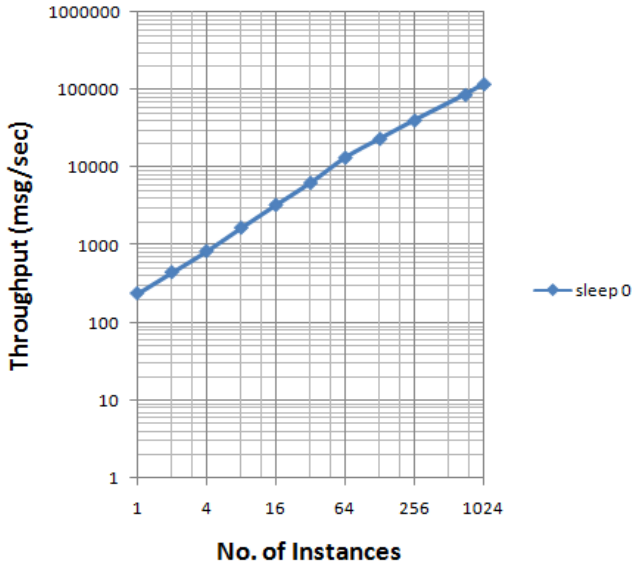

Figure 4 Throughput of CloudKonupto 1024 instances (MTC tasks)

C. Comparison with Matrix and Sparrow

We also got oppurtunity to work on CCGrid 2014 paper, where we published our throughput results. In the process, we also got to compare our results with 2 other job managements systems: Sparrow and MATRIX. Figure 5 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 MATRIX and Sparrow on 1 instances scale. The reason is that MATRIX runs locally without adding any scheduling or network overhead. But on CloudKon the tasks 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.

The throughput of MATRIX starts to decrease on larger scales. 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 crash. We were not able to run MATRIX on 256 instances. The network performance on EC2 cloud is much lower than then HPC clusters.

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. According to your suggestions we should try making some configuration changes in Sparrow to make it suitable to run on scales greater than 100.

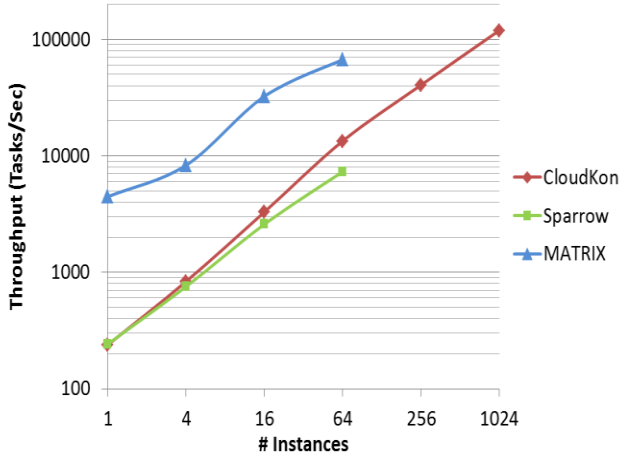

Figure 5. Throughput of CloudKon, Sparrow and MATRIX (MTC tasks)

D) Efficiency:

We tested the system efficiency in case of homogeneous tasks. The homogeneous tasks have a certain task duration length. Therefore it is easier 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.

In this section we evaluate the efficiency of CloudKon sub second tasks. It is important for sub-second task. Figure 8 shows the efficiency of 16 and 128 ms tasks on the systems. On sleep 16 ms tasks, the efficiency of CloudKon is around 40% which is low but is stable as the scale increases. That shows that CloudKon achieves a better scalability. On sleep 128 ms tasks, the efficiency of CloudKon is as high as 88%as shown in Fig. 6.

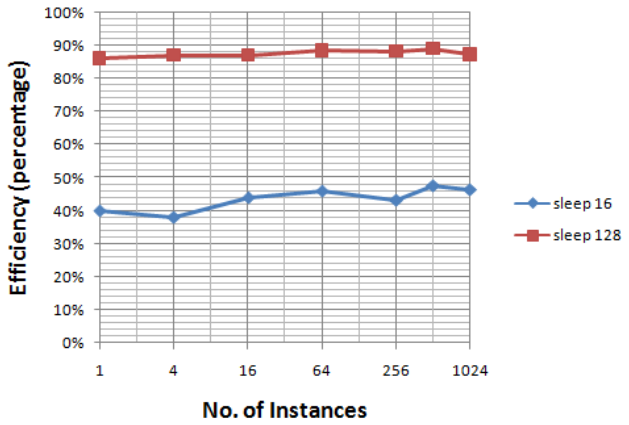

Figure 6. Efficiency of CloudKon running homogenous workloads

E) Consistency Overhead:

In this section we evaluate effect of tasks execution consistency on CloudKon. Figure 7 shows the system run-time for sleep 16 ms 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.

The results show that the overhead increases with the scale. The inconsistency on different scales comes from the fact that the number of the duplicate messages on each experiment could be different. That results in more random system performance of the system 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. As the number of instances increase, the probability of getting duplicate tasks becomes more. Therefore there will be more exceptions. That leads to a higher overhead. The overhead on larger scales goes up to %35 but it appears to be stable and not increasing.

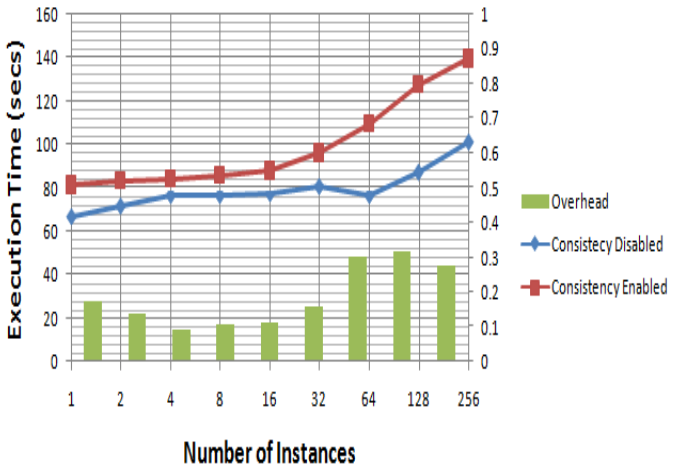

Figure 7. Consistency Overhead in CloudKon

Figure 8 also shows the throughput graph for forCloudKon with duplication and without duplication check. Both the throughput graphs are linear and increases with scale. The difference in gap between the duplication and without

duplication graph is increasing at lower scales but is constant at higher scales.

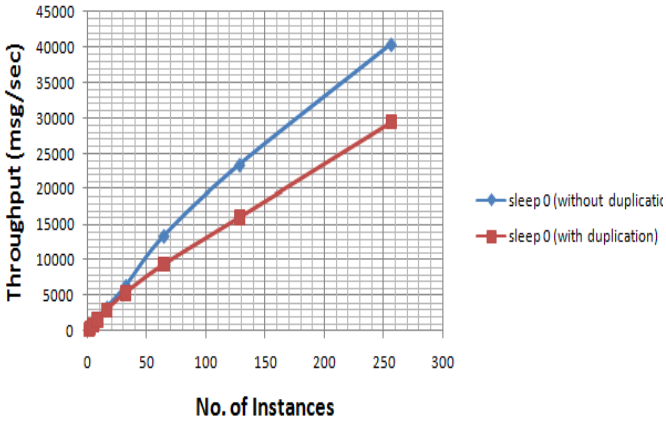

Figure 8. Throughput plot for Duplication and without Duplication

F) Utilization:

This section we evaluate the utilization of the worker threads for the intended task. The overall utilization is recorded and updated by Monitoring thread in Dynamo DB for every sec. We have evaluated using sleep 100 taskson 2 scales i.e. 4 and 8 nodes as shown if Fig. 9 and Fig. 10 respectively. The total no. of worker thread for 4 nodes = 4 * 4 i.e 16 worker threads. and for 8 nodes = 8 * 4 i.e. 32 worker threads.

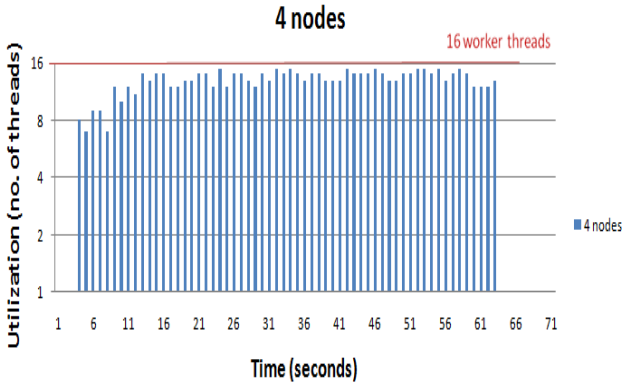

Figure 9. Average Utilization for sleep 100 on 4 node scale

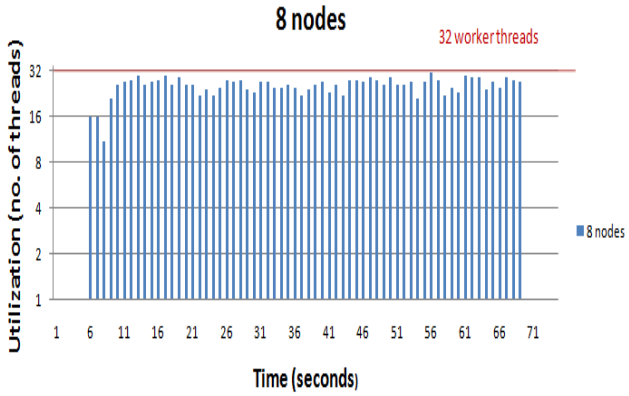

Figure 10. Average Utilization for sleep 100 on 8 node scale

The average utilization of the number of threads is around 13-14 threads/sec and 29-30 threads/sec respectively for 4 and 8 nodes as inferred from the Fig. 9 and 10. This shows that all the worker threads are effectively utilized during the system execution time.

G) Latency:

In order to measure latency accurately, the system has to record the request and respond timestamps of each task. Figure 11 shows the latency of CloudKon for sleep 0 ms scaling from 1 to 1024 instances. Each instance is running 2 client thread and 4 worker threads and sending 16000 tasks per instance. The latency of the system at 1 node is relatively high showing 3s overhead added by the system. But this will be acceptable when the scale increase. The latency is not too low because of the response bundling. The latency can be further reduced by adjusting the sleep time and bundling size on the server. The fact that the latency doesn't increase more than 5 s while increasing the scale from 1 instance to 1024 instance shows that CloudKon is stable. The main reason for that is that SQS as the task pool is a highly scalable service being backed up with multiple servers keeping the service very scalable. Therefore 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

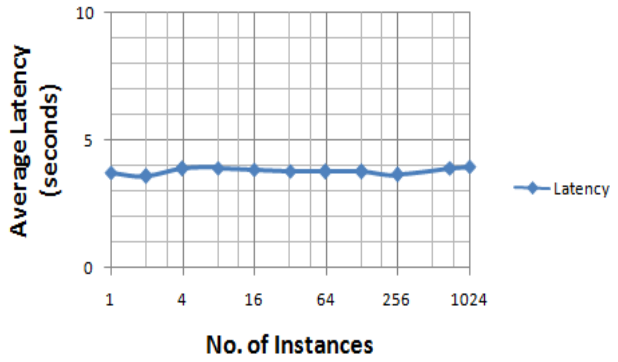

Figure 11. Latency of CloudKonsleep 0 ms tasks

IV. Related Work

The job schedulers could be centralized, where a single dispatcher manages the job submission, and job execution state updates; or hierarchical, where several dispatchers are organized in a tree-based topology; or distributed, where each computing node maintains its own job execution framework. In this section, we study commonly used examples of each type and point out their benefits and weaknesses compared to CloudKon [15].

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 at NASA Ames to address the needs of HPC, which is a highly configurable product that manages batch and interactive jobs, and adds the ability to signal, rerun and alter jobs. LSF Batch [20] 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) [21][22][23][24]. In [24], several distributed and hierarchical load balancing strategies are studied, such as Sender/Receiver Initiated Diffusion (SID/RID), Gradient Model (GM) and a Hierarchical Balancing Method (HBM). Other hierarchical strategies are explored in [23]. Charm++ [25] supports centralized, hierarchical and distributed load balancing. In [25], 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 [26]. 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.

Work stealing is another approach that has been used at small scales successfully in parallel languages such as Cilk [27], to load balance threads on shared memory parallel machines [28][29][31]. 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 [31]. The largest studies to date of work stealing have been at thousands of cores scales, showing good to excellent efficiency depending on the workloads [31].

To our knowledge CloudKon [15] is the only job management system along with Slurm++ that is able to support distributed HPC scheduling. It is able to run workloads of MTC, HPC or even workloads with combination those two. Moreover, CloudKon [15] is the only distributed task scheduler that is designed and optimized to run on public cloud environment. Slurm++ is a distributed job launch prototype, built on top of Slurm and ZHT (a distributed key value store) [32][33]. It supports job both HPC and MTC workloads. Slurm++ has been compared to SLURM up to 500 nodes and has shown 10X speedup.

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. Moreover, CloudKon [15] is the only distributed task scheduler that is designed and optimized to run on public cloud environment. CloudKon-Reloaded makes the maximum utilization of the processor with high level of concurrency, less network overhead and high efficiency.

V. Conclusion

We learned a lot of concepts about distributed system from this project. We were able to apply many theoretical concepts that we had learned in distributed system field. We became familiar with the Cloud computing building blocks (Amazon EC2, SQS, and DynamoDB) , problems in distributed system, multithreading, thread pool in java,developing a scalable and efficient code suitable for distributed system, debugging a highly distributed and multithreaded code, benchmarking at high scales, shell scripting, shell commands.

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.Our benchmarking results prove that the reloaded Cloudkon is highly scalable and provides high throughput, which also proves that 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. The comparison of CloudKon with other similar systems clearly shows that CloudKon was able to outperform other systems like Sparrow and MATRIX on scales of 128 instances or more in terms of throughput**.** From the Efficiency plot we can say that the code is about 87% efficient and the utilization of the system for the intended task is also very high as you can see from the utilization plot.

Future work can be done in many directions . One of the works would be to build a similar architecture outside cloud with same reliability, scalability and efficiency. With help from other systems such as ZHT Distributed Hash Table [32] [33] we will implement a SQS like queue in a way that can guarantee exactly once delivery. 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.

VI. References

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

[2] M. A. Jetteet. al, Slurm: Simple linux utility for resource management. In In Lecture Notes in Computer Sicence: Proceedings of Job Scheduling Strategies for PrarallelProcesing (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, 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)

[8] I. Raicu, et. al. "Toward Loosely Coupled Programming on Petascale Systems," IEEE SC 2008.

[9] I. Raicu, et. al. "Falkon: A Fast and Light-weight tasKexecutiON 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, R. S. Canon, K. Muriki, I. Sakrejda, and N. J. Wright. Evaluating Interconnect and virtualization performance for high performance computing, ACM Performance Evaluation Review, 40(2), 2012.

[12] PiyushMehrotra, JahedDjomehri, Steve Heistand, Robert Hood, Haoqiang Jin, Arthur Lazanoff, Subhash Saini, and Rupak Biswas. 2012. 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.

[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] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In IEEE INFOCOM, 2010.

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

[16] 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.

[17] 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

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

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

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

[21] L. V. Kal´eet. 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.

[22] 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.

[23] 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.

[24] 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

[25] 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.

[26] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: Scalable scheduling for sub-second parallel jobs. Tech. Rep. UCB/EECS-2013-29, EECS Department, University of California, Berkeley, Apr 2013.M.

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

[31] J. Dinanet. al. "Scalable work stealing," In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09), 2009.

[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, IEEE IPDPS '13, 2013.

[33] Tonglin Li, Raman Verma, Xi Duan, Hui Jin, IoanRaicu. "Exploring Distributed Hash Tables in High-End Computing", ACM Performance Evaluation Review (PER), 2011

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

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

[36] 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.

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

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