SCALABLE RESOURCE MANAGEMENT IN CLOUD COMPUTING

BY

IMAN SADOOGHI

DEPARTMENT OF COMPUTER SCIENCE

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Adviser

Chicago, Illinois
September 2016

ACKNOWLEDGEMENT

This dissertation could not have been completed without the consistent encouragement and support from my advisor Dr. Ioan Raicu and my family. There are many more people who help me succeed through my PhD study, particularly my dissertation committee: Dr. Zhiling Lan, Dr. Boris Glavic and Dr. Erdal Oruklu.

In addition, I would like to thank all my colleagues and collaborators for their time spent and efforts made in countless meetings, brainstorms, and running experiments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The exponential growth of data and application complexity has brought new challenges in the distributed computing field. Scientific applications are growing more diverse with various workloads, including traditional MPI high performance computing (HPC) to fine-grained loosely coupled many-task computing (MTC). Traditionally, these workloads have been shown to run well on supercomputers and highly-tuned HPC Clusters. The advent of Cloud computing has brought the attention of scientists to exploit these resources for scientific applications at a potentially lower cost. We investigate the nature of the cloud and its ability to run scientific applications efficiently. Delivering high throughput and low latency for the various types of workloads at large scales has driven us to design and implement new job scheduling and execution systems that are fully distributed and have the ability to run in public clouds. We discuss the design and implementation of a job scheduling and execution system (CloudKon) that has three major features: 1) it is optimized to exploit the cloud resources efficiently through a variety of cloud services (Amazon SQS and DynamoDB) in order to get the best performance and utilization; 2) it is fully distributed and it is able to run large scale applications; 3) it supports various workloads including MTC and HPC applications concurrently. To further improve the performance and the flexibility of CloudKon, we designed and implemented a fully distributed message queue (Fabriq) that delivers an order of magnitude better performance than the Amazon Simple Queuing System (SQS). Designing Fabriq helped us expand our scheduling system to many other distributed system including non-Amazon clouds. Having Fabriq as a building block, we were able to design and implement a multipurpose task scheduling and execution framework

(Albatross) that is able to efficiently run various types workloads at larger scales. Albatross provides data locality and task execution dependency. Those features enable Albatross to natively run MapReduce workloads. We evaluated CloudKon with synthetic MTC workloads, synthetic HPC workloads, and synthetic MapReduce applications on the Amazon AWS cloud with up to 1K instances. Fabriq was also evaluated with synthetic workloads on Amazon AWS cloud with up to 128 instances. Performance evaluations of Albatross show its ability to outperform Spark and Hadoop on different scenarios.

CHAPTER 1

INTRODUCTION

The advent of Big Data and the Exascale computing has changed many paradigms in the computing science area. More than 2.5 exabytes of data is generated every day, and more than 70% of it is unstructured 0. Various organizations including governments and big companies generate massive amounts of data in different formats including logs, and other unstructured raw data every day. Experts predict that by the end of 2018, the exascale (10^{18} FLOPS) computers will start to work [2]. Other predictions suggest that by the end of this decade, distributed systems will reach this scale with millions of nodes running billions of threads [3]. Many science domains (e.g. bio-informatics, drug discovery, energy, weather modeling, global warming, etc.) will achieve significant advancements due to exascale computing. However, running applications in such scales poses new scheduling and resource management challenges. One cannot expect to get satisfactory performance, efficiency and utilization by approaching the exascale systems with the traditional solutions. It is unlikely for the traditional centralized variations of the scheduling systems to be able to handle exascales [111]. Such systems are likely to get saturated at smaller scales. Therefore, there is emergent need for new scheduling systems that can provide acceptable performance on such scales without possessing significant overheads [110]. This has driven us to design and implement new job scheduling systems for the next generation distributed systems, specifically clouds. We have chosen to approach cloud environment as an alternative resource for scientific applications. Cloud

computing has gained the attention of scientists as a competitive resource to run HPC applications at a potentially lower cost. Thus, we have chosen to provide job scheduling solutions for large scale scientific computing on cloud environment. From this point, the terms of resource management system, resource manager, job scheduling system and job scheduler are used interchangeably. Also, the terms of job and task would be used interchangeably.

Traditionally, scientific applications have been shown to run well on supercomputers and highly-tuned HPC Clusters. Scientific applications usually require significant resources, however not all scientists have access to sufficient high-end computing systems. The idea of using clouds for scientific applications has been around for several years, but it has not gained traction primarily due to many issues such as lower network bandwidth or poor and unstable performance. Scientific applications often rely on access to large legacy data sets and pre-tuned application software libraries. These applications today run in HPC environments with low latency interconnect and rely on parallel file systems. They often require high performance systems that have high I/O and network bandwidth. Using commercial clouds gives scientists opportunity to use the larger resources on-demand. However, there is an uncertainty about the capability and performance of clouds to run scientific applications because of their different nature. Clouds have a heterogeneous infrastructure compared with homogenous high-end computing systems (e.g. supercomputers). The design goal of the clouds was to provide shared resources to multi-tenants and optimize the cost and efficiency. On the other hand, supercomputers are designed to optimize the performance and minimize latency. Before

choosing the cloud environment as an eligible competitive resource to run scientific applications, we need to assess its abilities and make sure it is capable to provide comparable performance. The first part of our research is to evaluate the capabilities of the cloud.

We chose Amazon AWS cloud as our main benchmarking target. The reason for this decision is that (1) it is the most commonly used public cloud (2) it is used by our job scheduling system, CloudKon. We first analyze the potentials of the cloud by evaluating the raw performance of different services of AWS such as compute, memory, network and I/O. Based on the findings on the raw performance, we then evaluate the performance of the scientific applications running in the cloud. Finally, we compare the performance of AWS with a private cloud, in order to find the root cause of its limitations while running scientific applications. We assess the ability of the cloud to perform well, as well as to evaluate the cost of the cloud in terms of both raw performance and scientific applications performance. Furthermore, we evaluate other services including S3, EBS and DynamoDB among many AWS services in order to assess the abilities of those to be used by scientific applications and frameworks. We also evaluate a real scientific computing application through the Swift parallel scripting system at scale.

Cloud computing has become a popular resource to host data-analytics workloads. Hadoop is a good data-analytics example for an application that could majorly benefit from running on cloud environments. Features such as reliability of the Hadoop framework enables it to fit well within the commodity resources of the cloud. However, exploiting the cloud resources efficiently at larger scales remains a major concern. There

is need for resource management and job scheduling systems that could manage cloud resources and distribute Hadoop jobs efficiently on those. This has been one of the main motivations of our work. The main goal of this work is to design and implement a distributed job scheduling system for scientific and data-analytics applications that could exploit the cloud environment resources efficiently, and also scale well on larger scales. The function 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. Unfortunately, today's schedulers have centralized Master/Slaves architecture (e.g. Slurm [4], Condor [5][6], PBS [7], SGE [8]), 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 [9], but it has poor scalability at the extreme scales of petascale systems with fine-granular workloads [10][11]. The solution to this problem is to move to the decentralized architectures that avoid using a single component as a manager. Distributed schedulers are usually implemented in either hierarchical [12] or fully distributed architectures [13] 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 [14].

Having extensive resources, public clouds could be exploited for executing tasks in extreme scales in a distributed fashion. In this project, we provide a compact and

lightweight distributed task execution framework that runs on the Amazon Elastic Compute Cloud (EC2) [15], by leveraging complex distributed building blocks such as the Amazon Simple Queuing Service (SQS) and the Amazon distributed NoSQL key/value store (DynamoDB) [17].

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. 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) [22]. 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 scheduler is a working evidence that shows 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. 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.

CloudKon is able to achieve great scalability while outperforming other state of the art scheduling systems like Sparrow [14]. However it has some limitations. Due to using SQS, CloudKon is locked down to Amazon EC2 cloud. That means users can only use it on AWS resources. That prevents us from testing our prototype on other environments such as other public/private cloud, or HPC resources. Moreover, due to running on a stand-alone separate server, SQS is not able to run internally on CloudKon. That adds significant overhead to the system that cannot be prevented. An open-sourced solution that could be integrated within the job scheduling system would suite it better. We investigated the available open-sourced options. The available options do not fit the CloudKon requirements well. Some queuing services add significant overhead to the system while others cannot scale well to large scales. In order to further improve the

performance and the flexibility of the CloudKon. That drove us to design and implement our own distributed message queue.

A Distributed Message Queue (DMQ) could be an important building block for a reliable distributed system. Message Queues could be useful in various data movement and communication scenarios. In High Throughput Computing (HTC), message queues can help decouple different components of a bigger system that aims to run in larger scales. Using distributed queues, different components can communicate without dealing with the blocking calls and tightly coupled communication.

We propose Fast, Balanced and Reliable Distributed Message Queue (Fabriq), a persistent reliable message queue that aims to achieve high throughput and low latency while keeping the near perfect load balance even on large scales. Fabriq uses ZHT as its building block. ZHT is a persistent distributed hash table that allows low latency operations and is able to scale up to more than 8k-nodes [23][107]. Fabriq leverages ZHT components to support persistence, consistency and reliable messaging.

Among the various DMQs, Fabriq and Kafka are the only alternatives that can provide the acceptable performance at larger scales required by CloudKon. Kafka is mainly optimized for large scale log delivery. It does not support multiple clients read from one broker at the same time. Moreover, it does not have a notion of independent messages or tasks. These limitations can significantly degrade the performance of CloudKon. Fabriq has none of those limitations. Leveraging Fabriq, CloudKon can run independently on any generic distributed system without being tied to SQS, DynamoDB, or the Amazon AWS Cloud in general. Moreover, our results show that Fabriq provides a

much higher throughput and much lower latency than SQS. According to our comparison results between SQS and Fabriq, and based on the fact that the future version of CloudKon will not have the overhead of DynamoDB, we expect about a 20X performance improvement (13X for using Fabriq and 1.5X for not using DynamoDB) on future version of CloudKon.

This work motivates the usage of the cloud environment for scientific applications. In order to assess the ability of cloud to run scientific applications, we design a methodology to evaluate the capabilities/ability of the cloud in both raw performance and the real applications performance. Then, we evaluate the performance of the Amazon AWS cloud as a pioneer public cloud.

After assessing the abilities of the cloud, we design and implement a distributed job scheduling system that runs on Amazon EC2. We propose CloudKon as a job management system that achieves good load balancing and high system utilization at large scales. Using CloudKon lets scientific applications exploit the distributed computing resources in any required scale in an on-demand fashion. Using cloud services such as Amazon SQS and DynamoDB that are integrated within the AWS software stack, our scheduler can optimally utilize cloud resources and achieve better performance. CloudKon uses a fully distributed queuing service (SQS) as its building block. 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 CloudKon 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 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%).

To further improve the performance and flexibility of CloudKon, we design and implement a distributed queuing service. We propose Fabriq, a distributed message queue that runs on top of a Distributed Hash Table. The design goal of Fabriq is to achieve lower latency and higher efficiency while being able to handle large scales. Moreover, Fabriq is persistent, reliable and consistent.

The results show that Fabriq was able to achieve high throughput in both small and large messages. At the scale of 128 nodes, Fabriq's throughput was as high as 1.8 Gigabytes/sec for 1 Megabytes messages, and more than 90,000 messages/sec for 50 bytes messages. At the same scale, Fabriq's latency was less than 1 millisecond. Our framework outperforms other state of the art systems including Kafka and SQS in throughput and latency. Furthermore, our experiments show that Fabriq provides a significantly better load balancing than Kafka. The load difference between Fabriq servers was less than 9.5% (compared to the even share), while in Kafka this difference was 100%, meaning that some servers did not receive any messages and remained idle.

The final goal of this thesis was to design and implement a distributed task scheduling and execution framework, that is able to efficiently run various types of workloads on cloud environment. We used Fabriq as a building block for our framework. Having a fully distributed architecture, Fabriq is the perfect fit to be used as a building block in our framework.

There is an emergent need for a fully distributed scheduler that handles the scheduling at the task level and is able to provide efficient scheduling for high granular tasks. In order to achieve scalability, it is important to avoid a centralized component as it could become a bottleneck. In this paper, we propose Albatross: A fully distributed cloud-enabled task scheduling and execution system that utilizes a distributed Message Queue as its building block.

The main idea of scheduling in Albatross is to use Fabriq, which is a distributed message queue, for delivering tasks to the workers in a parallel and scalable fashion. Most of the common schedulers have a central scheduler or a controller that distributes the tasks by pushing them to the worker nodes. However, unlike the traditional schedulers, Albatross uses a pulling approach as opposed to pushing tasks to the servers. The benefit of this approach is to avoid the bottleneck of having a regional or a central component for task distribution. Albatross also uses a Distributed Hash Table (DHT) for the metadata management of the workloads. There is no difference between any of the nodes in Albatross. Each node is a worker, a server, and possibly a client. The DMQ and the DHT are dispersed among all of the nodes in the system. The task submission, scheduling, and execution all happen through the collaboration of all of the system nodes. This feature enables Albatross to achieve a high scalability. The communication and the routing of the tasks all happen through hashing functions that have an O(1) routing complexity. That makes the communications between the servers optimal.

Albatross is able to run workflows with task execution dependency through a built in support in the DMQ. That gives Albatross flexibility to run HPC, and data analytics jobs.

An HPC job is usually defined as a Bag-of-Tasks [121] with dependencies between those tasks. The built-in task dependency support will enable the application to submit jobs to Albatross without having to provide an application level support for task dependencies. The Directed Acyclic Graph (DAG) support also enables Albatross to run various types of data analytics workloads. The focus of this paper is mainly Map-reduce workloads.

Another important feature that is required for data analytics frameworks is data locality support. Data locality suggests that since the movement of the data on the network between the nodes is an expensive process, the frameworks have to prioritize moving tasks to the data location and minimize the data movement on the system. In Albatross this feature is supported through load balancers of the Fabriq.

Our evaluations show that Albatross outperforms Spark and Hadoop that are currently state-of-the-art scheduling and execution frameworks for data analytics in many scenarios. It particularly outperforms the other two when the task granularity increases. Albatross is able to schedule tasks at 10K tasks per second rate, outperforming Spark by 10x. The latency of Albatross is almost an order of magnitude lower than Spark. Albatross's throughput on real applications has been faster than the two other systems by 2.1x and 12.2x. Finally, it outperforms Spark and Hadoop respectively by 46x, and 600x in processing high granularity workloads on grep application.

In summary, the main contributions of this work are as follows:

(1) A comprehensive study on scientific applications characteristics and evaluation of their performance on clouds. The study analyzes the potentials of the cloud as an alternative environment for scientific computing [102].

(2) A distributed job scheduling system (CloudKon) design that suites the cloud's characteristics. A system that is able to support HPC and MTC workloads. We conduct a performance evaluation up to 1024 instances scale. [46]

(3) A distributed message queuing (Fabriq) system that is scalable and provides ultra low latency. Fabriq exploits distributed hash tables as a building block to deliver a highly scalable solution. The proposed system is able to achieve near perfect load balancing and sub-milliseconds distribution latency. Fabriq offers support for substantial features such as persistence, consistency, reliability, dynamic scalability, and message delivery guarantees. [103]

(4) An advanced vendor-independent distributed task scheduling and job execution framework (Albatross) that suites cloud environment. The proposed system provides native support for task execution dependency and data locality. Albatross is able to run tasks with the granularity of sub-milliseconds efficiently.

CHAPTER 2

UNDERSTANDING THE PERFORMANCE AND POTENTIAL OF CLOUD COMPUTING FOR SCIENTIFIC APPLICATIONS

As we explained previously, before choosing to exploit the public cloud for scientific computing, we need to assess its abilities in different aspects. In this chapter, we provide a comprehensive evaluation of EC2 cloud in different aspects.

2.1 Background and Motivation

Commercial clouds bring a great opportunity to the scientific computing area. Scientific applications usually require significant resources, however not all scientists have access to sufficient high-end computing systems. Cloud computing has gained the attention of scientists as a competitive resource to run HPC applications at a potentially lower cost. But as a different infrastructure, it is unclear whether clouds are capable of running scientific applications with a reasonable performance per money spent. Moreover, clouds are usually comprised of heterogeneous resources as opposed to the homogenous HPC resources. The architecture of the cloud is optimized to provide resource sharing among various users. On the other hand, supercomputers were designed to provide dedicated resources with optimum performance and minimum latency.

Clouds have some benefits over supercomputers. They offer more flexibility in their environment. Scientific applications often have dependencies on unique libraries and platforms. It is difficult to run these applications on supercomputers that have shared resources with pre-determined software stack and platform, while cloud environments

also have the ability to set up a customized virtual machine image with specific platform and user libraries. This makes it very easy for legacy applications that require certain specifications to be able to run. Setting up cloud environments is significantly easier compared to supercomputers, as users often only need to set up a virtual machine once and deploy it on multiple instances. Furthermore, with virtual machines, users have no issues with custom kernels and root permissions (within the virtual machine), both significant issues in non-virtualized high-end computing systems.

There are some other issues with clouds that make them challenging to be used for scientific computing. The network bandwidth in commercial clouds is significantly lower (and less predictable) than what is available in supercomputers. Network bandwidth and latency are two of the major issues that cloud environments have for high-performance computing. Most of the cloud resources use commodity network with significantly lower bandwidth than supercomputers [33].

The virtualization overhead is also another issue that leads to variable compute and memory performance. I/O is yet another factor that has been one of the main issues on application performance. Over the last decade the compute performance of cutting edge systems has improved in much faster speed than their storage and I/O performance. I/O on parallel computers has always been slow compared with computation and communication. This remains to be an issue for the cloud environment as well.

Finally, the performance of parallel systems including networked storage systems such as Amazon S3 needs to be evaluated in order to verify if they are capable of running scientific applications. All of the above mentioned issues raise uncertainty for the ability

of clouds to effectively support HPC applications. Thus it is important to study the capability and performance of clouds in support of scientific applications. Although there have been early endeavors in this aspect [19][34][21][38][40], we develop a more comprehensive set of evaluation. In some of these works, the experiments were mostly run on limited types and number of instances [34][21][35]. Only a few of the researches have used the new Amazon EC2 cluster instances that we have tested [19][38][41]. However the performance metrics in those works are very limited. This chapter covers a thorough evaluation covering major performance metrics and compares a much larger set of EC2 instance types and the commonly used Amazon Cloud Services. Most of the aforementioned above mentioned works lack the cost evaluation and analysis of the cloud. Our work analyses the cost of the cloud on different instance types.

The main goal of this chapter is to evaluate the performance of the Amazon public cloud as the most popular commercial cloud available, as well as to ***offer some context for comparison against a private cloud solution***. We run micro benchmarks and real applications on Amazon AWS to evaluate its performance on critical metrics including throughput, bandwidth and latency of processor, network, memory and storage [15]. Then, we evaluate the performance of HPC applications on EC2 and compare it with a private cloud solution [29]. This way we will be able to better identify the advantages and limitations of AWS on the scientific computing area.

Over the past few years, some of the scientific frameworks and applications have approached using cloud services as their building blocks to alleviate their computation

processes [32][46]. We evaluate the performance of some of the AWS services such as S3 and DynamoDB to investigate their abilities on scientific computing area.

Finally, this work performs a detailed price/cost analysis of cloud instances to better understand the upper and lower bounds of cloud costs. Armed with both detailed benchmarks to gauge expected performance and a detailed monetary cost analysis, we expect ***this chapter will be a recipe cookbook for scientists to help them decide where to deploy and run their scientific applications between public clouds, private clouds, or hybrid clouds***.

The rest of this chapter is organized as follows: Section 2.2 provides the evaluation of the EC2, S3 and DynamoDB performance on different service alternatives of Amazon AWS. We provide an evaluation methodology. Then we present the benchmarking tools and the environment settings of the testbed in this project. Section 2.2.4 presents the benchmarking results and analyzes the performance. On 2.2.5 we compare the performance of EC2 with FermiCloud on HPL application. Section 2.3 analyzes the cost of the EC2 cloud based on its performance on different aspects. Section 2.4 summarizes this chapter and discusses future work.

2.2 Performance Evaluation

In this section we provide a comprehensive evaluation of the Amazon AWS technologies. We evaluate the performance of Amazon EC2 and storage services such as S3 and EBS. We also compare the Amazon AWS public cloud to the FermiCloud private cloud.

2.2.1 Methodology

We design a performance evaluation method to measure the capability of different instance types of Amazon EC2 cloud and to evaluate the cost of cloud computing for scientific computing. As mentioned, the goal is to evaluate the performance of the EC2 on scientific applications. To achieve this goal, we first measure the raw performance of EC2. We run micro benchmarks to measure the raw performance of different instance types, compared with the theoretical performance peak claimed by the resource provider. We also compare the actual performance with a typical non-virtualized system to better understand the effect of virtualization. Having the raw performance we will be able to predict the performance of different applications based on their requirements on different metrics. Then we compare the performance of a virtual cluster of multiple instances running HPL application on both Amazon EC2 and the FermiCloud. Comparing the performance of EC2, which we do not have much information about its underlying resources with the FermiCloud, which we know the details about, we will be able to come up with a better conclusion about the weaknesses of the EC2. On the following sections we try to evaluate the performance of the other popular services of Amazon AWS by comparing them to the similar open source services.

Finally, we analyze the cost of the cloud computing based on different performance metrics from the previous part. Using the actual performance results provides more accurate analysis of the cost of cloud computing while being used in different scenarios and for different purposes.

The performance metrics for the experiments are based on the critical requirements of scientific applications. Different scientific applications have different priorities. We need to know about the compute performance of the instances in case of running compute intensive applications. We also need to measure the memory performance, as memory is usually being heavily used by scientific applications. We also measure the network performance which is an important factor on the performance of scientific applications.

2.2.2 Benchmarking tools and applications

It is important for us to use wide-spread benchmarking tools that are used by the scientific community. Specifically in Cloud Computing area, the benchmarks should have the ability to run over multiple machines and provide accurate aggregate results.

For memory we use CacheBench. We perform read and write benchmarks on single instances. For network bandwidth, we use Iperf [26]. For network latency and hop distance between the instances, we use ping and traceroute. For CPU benchmarking we have chosen HPL benchmark [27]. It provides the results in floating-point operations per second (FLOPS).

In order to benchmark S3, we had to develop our own benchmark suite, since none of the widespread benchmarking tools can be used to test storage like this. We have also developed a tool for configuring a fully working virtual cluster with support for some specific file systems.

2.2.3 Parameter space and testbed

In order to better show the capability of Amazon EC2 on running scientific applications we have used two different cloud infrastructures: (1) Amazon AWS Cloud,

and (2) FermiCloud. Amazon AWS is a public cloud with many datacenters all around the world. FermiCloud is a private Cloud which is used for internal use in Fermi National Laboratory.

In order to compare the virtualization effect on the performance we have also included two local systems on our tests: (1) A 6-core CPU and 16 Gigabytes of memory system (DataSys), and (2) a 48-cores and 256 Gigabytes memory system (Fusion).

a. Amazon EC2

The experiments were executed on three Amazon cloud data centers: US East (Northern Virginia), US West (Oregon) and US West (Northern California). We cover all of the different instance types in our evaluations.

The operating system on all of the US West instances and the local systems is a 64bits distribution of Ubuntu. The US East instances use 64 bits CentOS operating system. The US West instances use Para-virtualization technique on their hypervisor. But the HPC instances on the US East cloud center use Hardware-Assisted Virtualization (HVM) [29]. HVM techniques use the features of the new hardware to avoid handling all of the virtualization tasks like context switching or providing direct access to different devices at the software level. Using HVM, Virtual Machines can have direct access to hardware with the minimal overhead.

We have included different instances as well as a non-virtualized machine. The m1.small instance is a single core instance with low compute and network performance. M1.medium is a single core system with 3.75 GB of memory. C1.xlarge instance is a compute optimized with 8 cores and 7 GB of memory. M2.4xlarge is a memory

optimized instances and is supposed to have high memory performance. Hi1.4xlarge is a storage optimized instace with 2 SSD drives. Finally cc1.4xlarge and cc2.8xlarge as cluster compute instances, and c3.8xlarge as the new generation of HPC instances have 16 and 32 cores and more than 40 GB memory. These instances are optimized for HPC workloads.

b. FermiCloud

FermiCloud is a private cloud providing Infrastructure-as-a-Service services internal use. It manages dynamically allocated services for both interactive and batch processing. As part of a national laboratory, one of the main goals FermiCloud is being able to run scientific applications and models. FermiCloud uses OpenNebula Cloud Manager for the purpose of managing and launching the Virtual Machines [43]. It uses KVM hypervisor that uses both para-virtualization and full virtualization techniques [48]. The FermiCloud Infrastructure is enabled with 4X DDR Infiniband network adapters. The main challenge to overcome in the deployment of the network is introduced when virtualizing the hardware of a machine to be used (and shared) by the VMs. This overhead slows drastically the data rate reducing the efficiency of using a faster technology like Infiniband. To overcome the virtualization overhead they use a technique called Single Root Input/output Virtualization (SRIOV) that achieves device virtualization without using device emulation by enabling a device to be shared by multiple virtual machines. The technique involves with modifications to the Linux's Hypervisor as well as the OpenNebula manager [47].

Each server is enabled with a 4x (4 links) Infiniband card with a DDR data rate for a total theoretical speed of up to 20 Gb/s and after the 8b/10b codification 16 Gb/s. Network latency is 1 μs when used with MPI [28]. Each card has 8 virtual lanes that can create 1 physical function and 7 virtual functions via SR-IOV. The servers are enabled with 2 quad core 2.66 GHz Intel processors, 48Gb of RAM and 600Gb of SAS Disk, 12TB of SATA, and 8 port RAID Controller [47].

c. Performance Evaluation of AWS Memory hierarchy performance

This section presents the memory benchmark results. We sufficed to run read and write benchmarks. The experiments for each instance were repeated three times.

Memory bandwidth is a critical factor in scientific applications performance. Many Scientific applications like GAMESS, IMPACT-T and MILC are very sensitive to memory bandwidth [30]. Amazon has not included the memory bandwidth of the instances. It has only listed their memory size. We also measure the memory bandwidth of each instance.

Figure 1 shows the system memory read bandwidth in different memory hierarchy levels. The horizontal axis shows the cache size. The bandwidth is very stable up to a certain cache size. The bandwidth starts to drop after a certain size. The reason for the drop off is surpassing the memory cache size at a certain hierarchy level.

Memory performance of the m1.small instance is significantly lower than other instances. The low memory bandwidth cannot be only attributed to the virtualization overhead. We believe the main reason is memory throttling imposed based on the SLA of those instances.

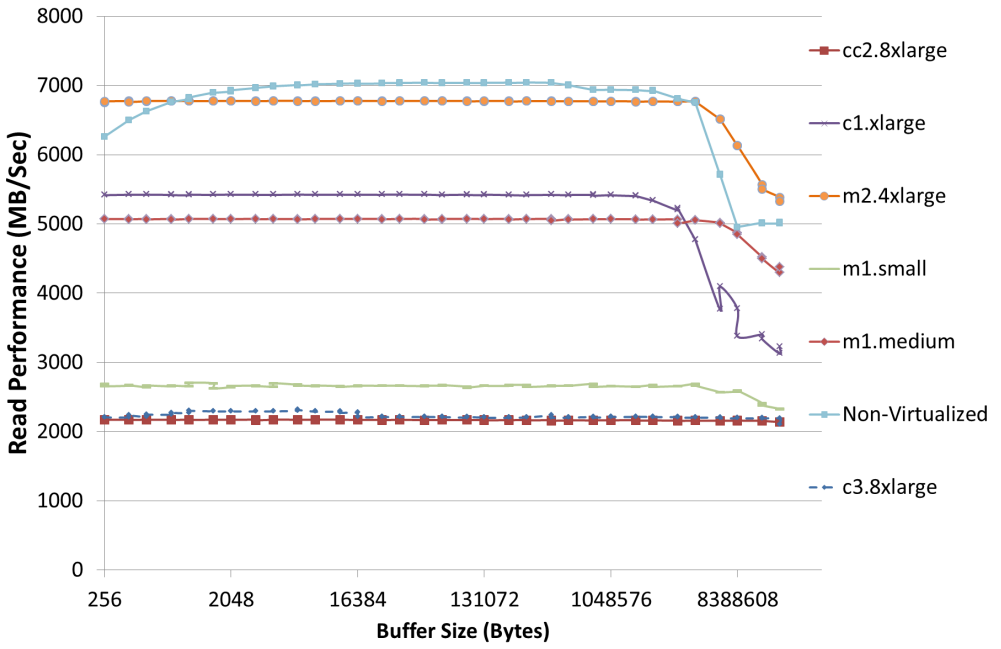

Figure 1. CacheBench Read benchmark results, one benchmark process per instance

Another noticeable point is the low bandwidth of the cc2.8xlarge and c3.8xlarge. These instances have similar performance that is much lower than other instances. A reason for that can be the result of the different virtual memory allocation on the VMs by HVM virtualization on these instances. We have however observed an effect in large hardware-assisted virtual machines such as those on FermiCloud. In such machines, it will take a while for the system to balance the memory out to its full size at the first launch of the VM.

After all, the results show that the memory bandwidth for read operation in the larger instances is close to the local non-virtualized system. *We can conclude that the*

virtualization effect on the memory is low, which is a good sign for scientific applications that are mostly sensitive to the memory performance.

Figure 2 shows the write performance of different cloud instances and the local system. The write performance shows different results from the read benchmark. As in write, the c3.8xlarge instance has the best performance next to the non-virtualized local system.

For each instance we can notice two or three major drop-offs in bandwidth. These drop-offs show different memory hierarchies. For example on the c3.8xlarge instance we can notice that the memory bandwidth drops at 24 Kbytes. We can also observe that the write throughputs for different memory hierarchies are different. These data points likely represent the different caches on the processor (e.g. L1, L2, L3 caches).

Comparing the cluster instance with the local system, we observe that on smaller buffer sizes, the local system performs better. But cloud instance outperforms the local system on larger cache sizes. The reason for that could be the cloud instances residing on more powerful physical nodes with higher bandwidths. We can observe that the write bandwidth on the cloud instances drops off at certain buffer sizes. That shows the memory hierarchy effects on the write operation.

Users can choose the best transfer size for write operation based on the performance peaks of each instance type to get the best performance. This would optimize a scientific application write bandwidth.

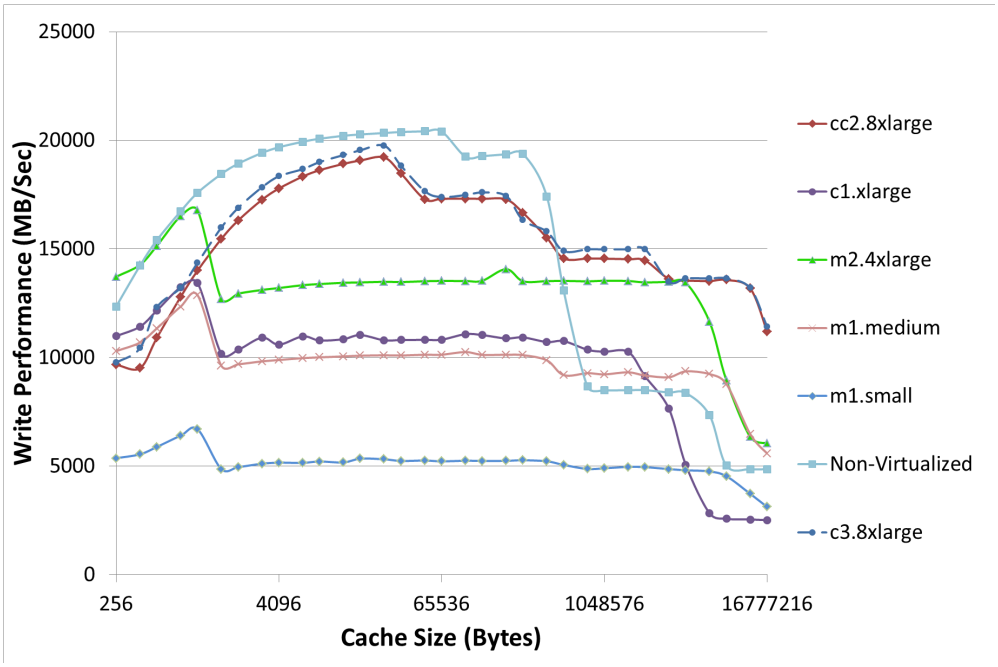

Figure 2. CacheBench write benchmark results, one benchmark process per instance

d. Network performance

We have run many experiments on network performance of Amazon cloud. The experiments test the network performance including bandwidth and latency.

We first test the local network bandwidth between the same types of instances. Figure 3 shows the network performance of different types of nodes. In each case both of the instances were inside the same datacenter. The network bandwidth for most of the instances were as expected except for two instances.

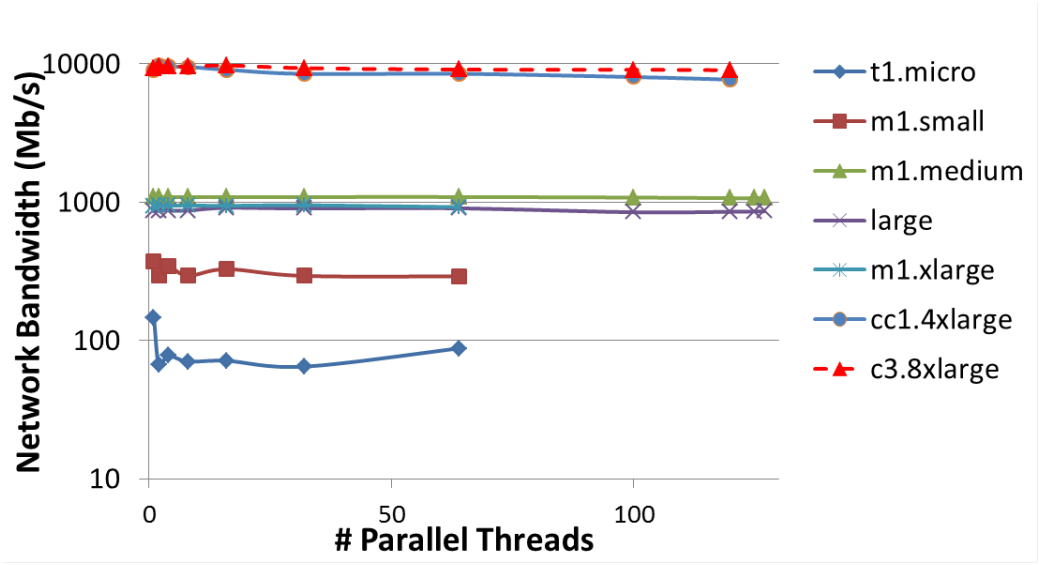

Figure 3. iPerf: Network bandwidth in a single client and server connection.

The lowest performance belongs to the t1.micro and m1.small instances. These two instances use the same 1 Gb/s network cards used by other instances. But they have much lower bandwidth. We believe that the reason is sharing the CPU cores and not having a dedicated core. This can affect network performance significantly as the CPU is shared and many network requests cannot be handled while the instance is on its idle time. During the idle time of the instance, the virtual system calls to the VMM will not be processed and will be saved in the queue until the idle time is over. The network performance is highly affected by processor sharing techniques. Other works had the same observations and conclusions about the network performance in these two instance types [18]. Another reason for the low performance of the m1.small and t1.micro instances could be throttling the network bandwidth by EC2. The Xen hypervisor has the ability of network throttling if needed.

Among the instances that use the slower network cards the m1.medium instance has the best performance. We did not find a technical reason for that. The m1.medium instances use the same network card as other instances and does not have any advantage on system configuration over other instance types. We assume the reason for that is the administrative decision on hypervisor level due to their popularity among different instance types.

Another odd result is for m1.medium instance. The bandwidth in medium instance exceeds 1 Gb/Sec, which is the specified network bandwidth of these. m1.medium instance bandwidth achieves up to 1.09 Gb/sec. That is theoretically not possible for a connection between two physical nodes with 1 Gb/s network cards. We believe the reason is that both of the VMs reside in the same physical node or the same cluster. In case of residing on the same node, the packets stay in the memory. Therefore the connection bandwidth is not limited to the network bandwidth. We can also assume that not necessarily the instances have 1 Gb/s network cards. In fact the nodes that run medium instances may have more powerful network cards in order to provide better network performance for these popular instances.

The HPC instances have the best network bandwidth among the instances. They use 10 Gb/sec network switches. The results show that the network virtualization overhead in these instances is very low. The performance gets as high as 97% of ideal performance.

We also measure the network connection latency and the hop distance between instances inside the Oregon datacenter of Amazon EC2. We run this experiment to find out about the correlation of connection latency and the hop distance. We also want to find

the connection latency range inside a datacenter. We measure the latency and the hop distance on 1225 combinations of m1.small instances. Figure 4 shows the network latency distribution of EC2 m1.small instances. It also plots the hop distance of two instances. The network latency in this experiment varies between 0.006 ms and 394 ms, an arguably very large variation.

We can observe from the results that: (1) 99% of the instances which have the transmission latency of 0.24 to 0.99 ms are 4 or 6 hops far from each other. So we can claim that if the latency is between 0.24 to 0.99 ms the distance between the instances is 4 to 6 hops with the probability of 99%. (2) More than 94% of the allocated instances to a user are 4-6 percent far from each other. In other words the hop distance is 4-6 instances with the probability of more than 94%.

We can predict the connection latency based on the hop distance of instances. We have run the latency test for other instance types. The results do not seem to be dependent on instance type for the instances with the same network interconnect. *The latency variance of Amazon instances is much higher than the variance in a HPC system. The high latency variance is not desirable for scientific applications. In case of HPC instances which have the 10 Gigabit Ethernet cards, the latency ranges from 0.19ms to 0.255ms which shows a smaller variance and more stable network performance.*

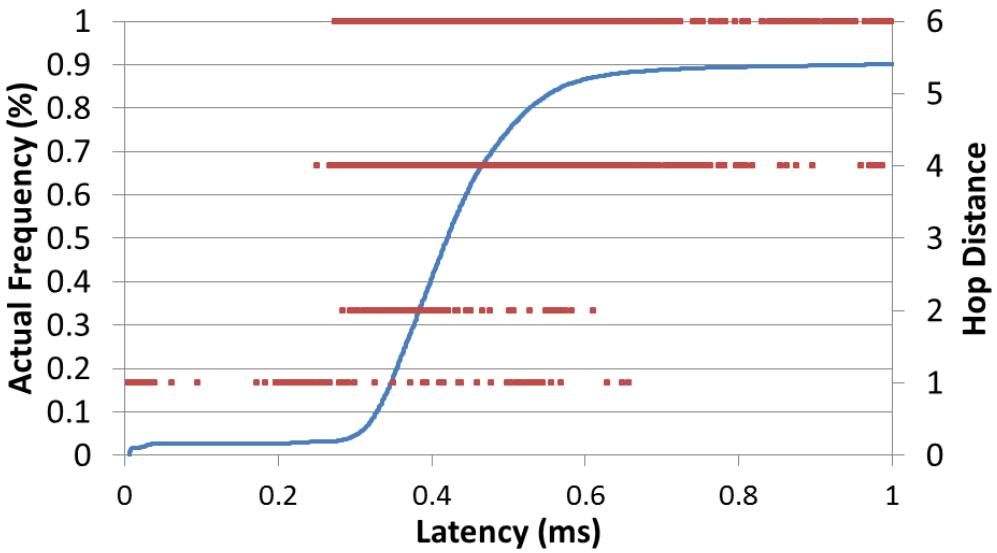

Figure 4. Cumulative Distribution Function and Hop distance of connection latency between instances inside a datacenter.

Other researches have compared the latency of EC2 HPC instances with HPC systems. The latency of the HPC instance on EC2 is reported to be 3 to 40 times higher than a HPC machine with a 23 Gb/s network card [19]. The latency variance is also much higher.

e. Compute Performance

In this section we evaluate the compute performance of EC2 instances. Figure 5 shows the compute performance of each instance using HPL as well as the ideal performance claimed by Amazon. It also shows the performance variance of instances.

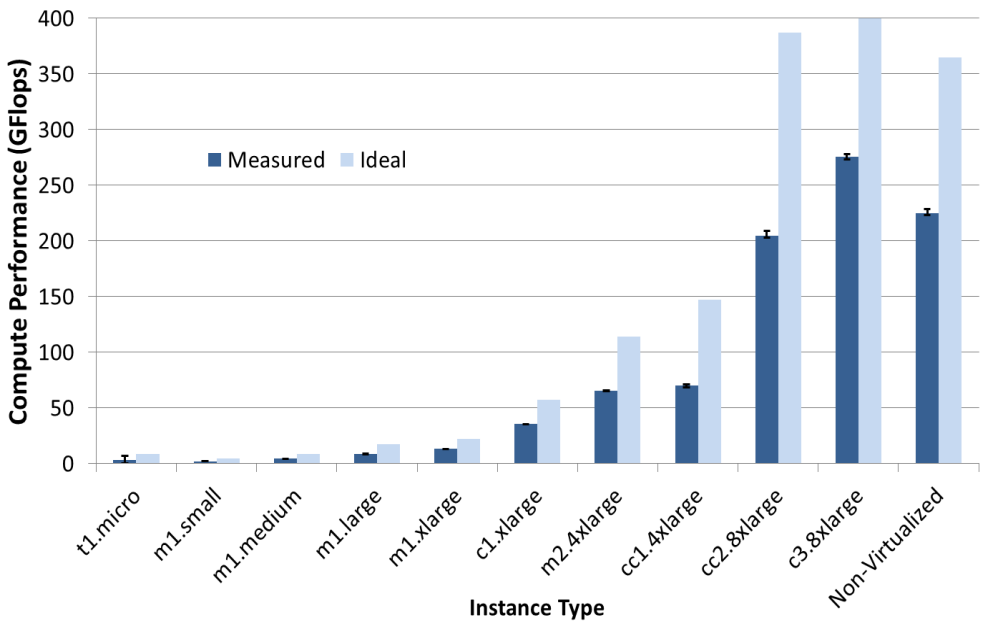

Figure 5. HPL benchmark results: compute performance of single instances comparing with their ideal performance.

Among the Amazon instances, the c3.8xlarge has the best compute performance. The t1.micro instance shows the lowest performance. The figure also shows the performance variance for each instance. The performance variance of the instances is low in most of the instance types. Providing a consistent performance is an advantage for cloud instances.

Among all of the instances, the c3.8xlarge and the non-virtualized node achieve the best efficiency. Overall we can observe that the efficiency of non-HPC instances is relatively low. Other papers have suggested the low performance of HPL application while running on virtualized environments [31][34]. However, noticing the fact that the HPC instances were as efficient as the non-virtualized node, and the fact that there is no

other factor (e.g. network latency) affecting the benchmark, can imply that the virtualization overhead has no major effect on this program on a single node scale.

f. I/O Performance

In this section we evaluate the I/O performance of the EBS volume and local storage of each instance. The following charts show the results obtained after running IOR on the local storage and EBS volume storage of each of the instances with different transfer sizes and storage devices. Figure 6 shows the performance of POSIX read operation on different instances. Except for the hi1.4xlarge, which is equipped with SSDs, the throughput among other instances does not vary greatly from one another. For most of the instances the throughput is close to a non-virtualized system with a normal spinning HDD.

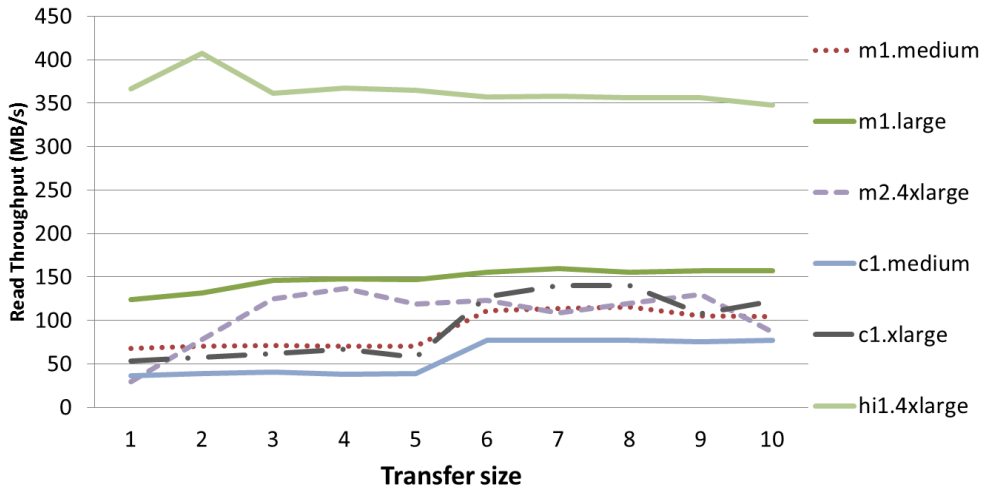

Figure 6. Local POSIX read benchmark results on all instances

Figure 7 shows the maximum write and read throughput on each instance on both EBS volumes and local storage devices. Comparing with local storage, EBS volumes

show a very poor performance, which is the result of the remote access delay over the network.

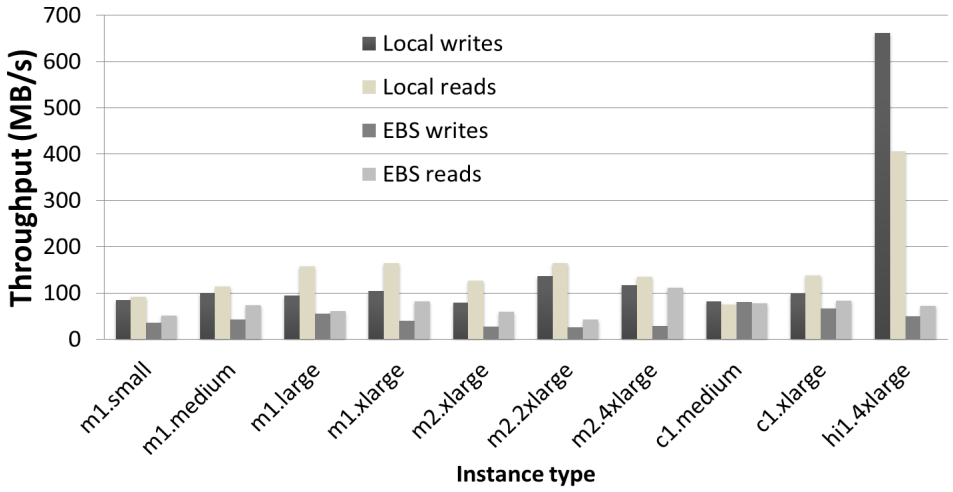

Figure 7. Maximum write/read throughput on different instances

Finally, to complete these micro-benchmarks, we set up a software RAID-0 with EBS volumes, varying the number of volumes from 1 to 8. We ran the same benchmark on a c1.medium instance. Figure 8 shows the write performance on RAID-0 on different number of EBS volumes. Looking at the write throughput, we can observe that the throughput does not vary a lot and is almost constant as the transfer size increases. That shows a stable write throughput on EBS drives. The write throughput on the RAID-0 increases with the number of drives. The reason for that is that the data will be spread among the drives and is written in parallel to all of the drives. That increases the write throughput because of having parallel write instead of serial write. Oddly, the performance does not improve as the number of drives increases from 1 to 2 drives. The reason for that is moving from the local writes to network. Therefore the throughput stays

the same. For 4 EBS volumes, we can observe a 4x increase on the throughput. In case of 8 EBS volumes we expect a 2x speed up comparing with the 4 EBS experiment. However the write throughput cannot scale better because of the limitation of the network bandwidth. The maximum achievable throughput is around 120MB/s, which is bound to the network bandwidth of the instances that is 1 Gb/s. so we can conclude that the RAID throughput will not exceed 120 MB/s if we add more EBS volumes.

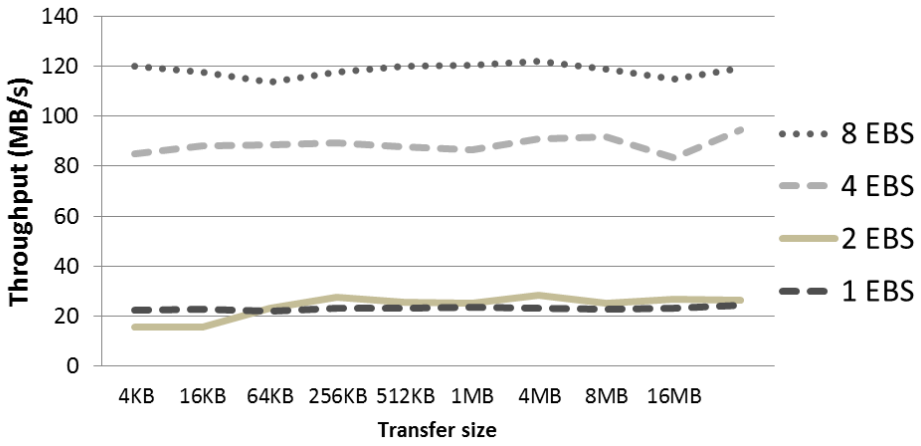

Figure 8. RAID0 Setup benchmark for different transfer sizes – write

g. S3 and PVFS Performance

In this section we evaluate and compare the performance of S3 and PVFS. S3 is a highly scalable storage service from Amazon that could be used on multinode applications. Also, a very important requirement for most of the scientific applications is a parallel file system shared among all of the computing nodes. We have also included the NFS as a centralized file system to show how it performs on smaller scales.

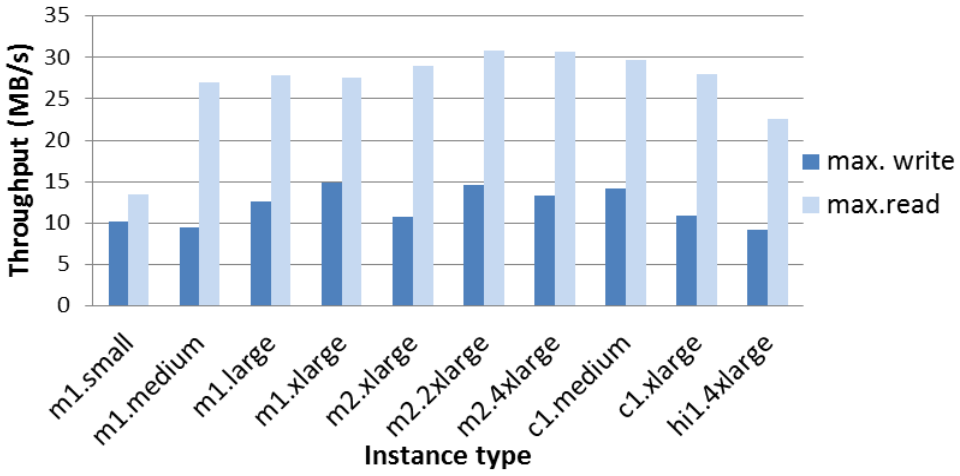

Figure 9. S3 performance, maximum read and write throughput

First we evaluate the s3 performance on read and write operations. Figure 9 shows the maximum read and write throughput on S3 accessed by different instance types. Leaving aside the small instances, there is not much difference between the maximum read/write throughput across instances. The reason is that these values are implicitly limited by either the network capabilities or S3 itself.

Next, We compare the performance of the S3 and PVFS as two possible options to use for scientific applications. PVFS is commonly used in scientific applications on HPC environments. On the other hand, S3 is commonly used on the multi-node applications that run on cloud environment. We have only included the read performance in this chapter. The experiment runs on m1.medium instances. Figure 10 shows that the read throughput of the S3 is much lower compared to PVFS on small scales. This results from the fact that the S3 is a remote network storage while PVFS is installed and is spread over each instance. As The number of the instances increase, PVFS cannot scale as well as the

S3 and the performance of the two systems get closer to each other up to a scale that S3 slightly performs better than the PVFS. Therefore it is better to choose S3 if we are using more than 96 instances for the application.

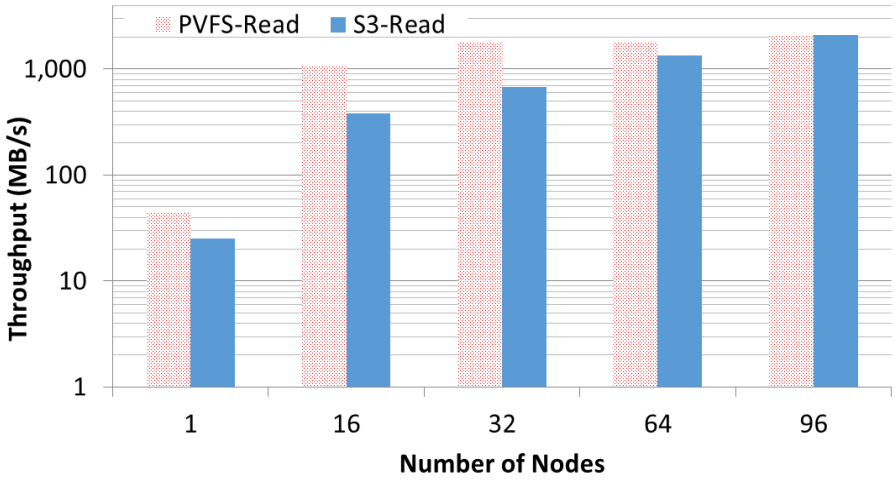

Figure 10. Comparing the read throughput of S3 and PVFS on different scales

Next, we evaluate the performance of PVFS2 for the scales of 1 to 64 as we found out that it performs better than S3 in smaller scales. To benchmark PVFS2 for the following experiments we use the MPIIO interface instead of POSIX. In the configuration that we used, every node in the cluster serves both as an I/O and metadata server. Figure 11 shows the read operation throughput of PVFS2 on local storage with different number of instances and variable transfer size. The effect of having a small transfer size is significant, where we see that the throughput increases as we make the transfer size bigger. Again, this fact is due to the overhead added by the I/O transaction.

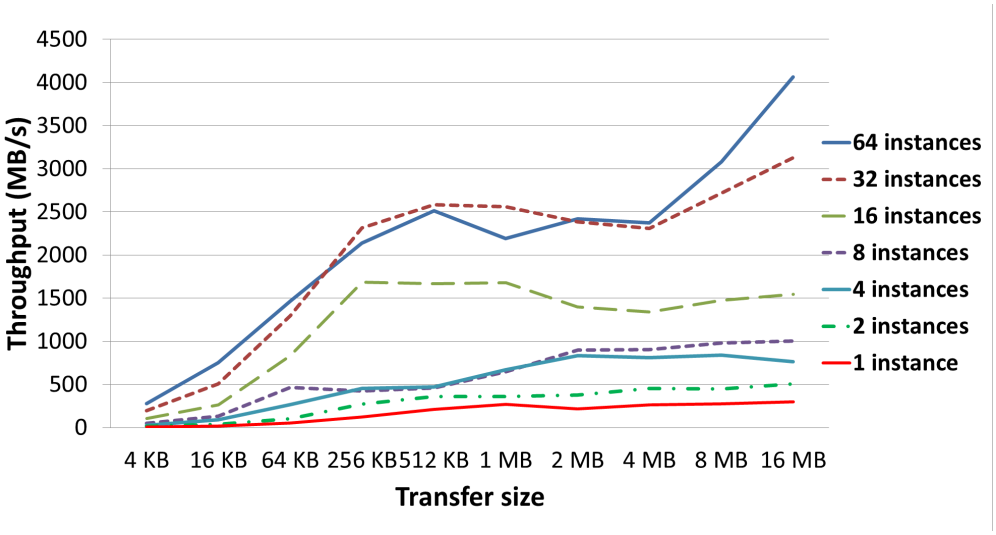

Figure 11. PVFS read on different transfer sizes over instance storage

Finally, Figure 12, shows the performance of PVFS2 and NFS on memory through the POSIX interface. The results show that the NFS cluster does not scale very well and the throughput does not increase as we increase the number of nodes. It basically bottlenecks at the 1Gb/s which is the network bandwidth of a single instance. PVFS2 performs better as it can scale very well on 64 nodes on memory. But as we have shown above, it will not scale on larger scales.

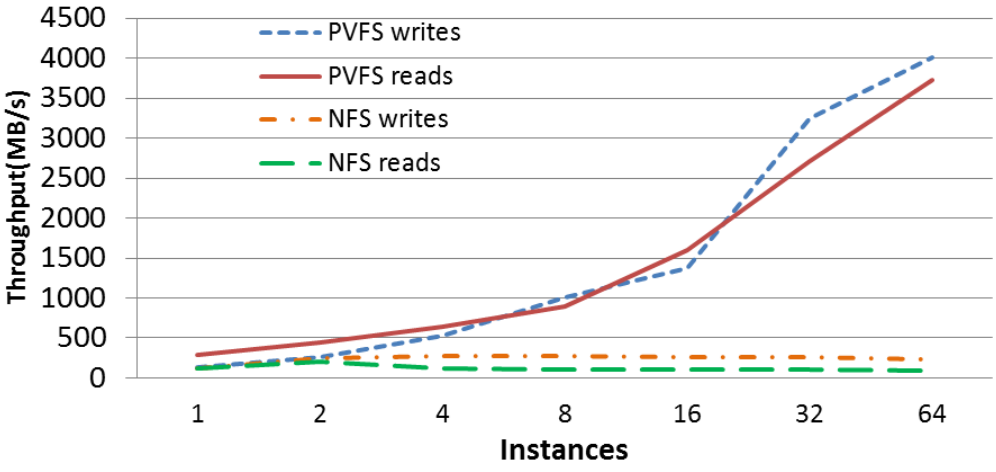

Figure 12. Scalability of PVFS2 and NFS in read/write throughput using memory as storage

h. DynamoDB performance

In this section we are evaluating the performance of Amazon DynamoDB. DynamoDB is a commonly used NoSql database used by commercial and scientific applications [17]. We conduct micro benchmarks to measure the throughput and latency of insert and look up calls scaling from 1 to 96 instances with total number of calls scaling from 10000 to 960000 calls. We conduct the benchmarks on both m1.medium and cc2.8xlarge instances. The provision capacity for the benchmarks is 10K operations/s which is the maximum default capacity available. There is no information released about how many nodes are used to offer a specific throughput. We have observed that the latency of DynamoDB doesn't change much with scales, and the value is around 10ms. This shows that DynamoDB is highly scalable. **Figure 13** shows the latency of look up and insert calls made from 96 cc2.8xxlarge instances. The average latency for insert and look up are respectively 10 ms and 8.7 ms. 90% of the calls had a latency of less than 12 ms for insert and 10.5 ms for look up.

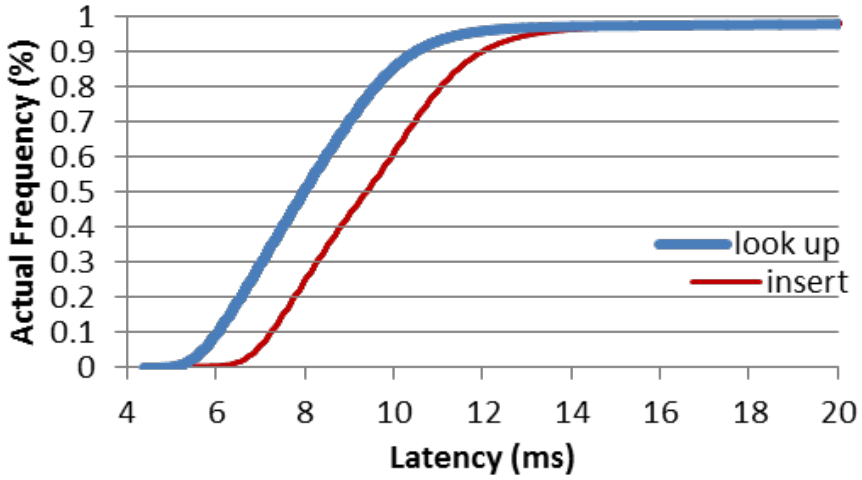

Figure 13. CDF plot for insert and look up latency on 96 8xxl instances

We compare the throughput of DynamoDB with ZHT on EC2 [107]. ZHT is an open source consistent NoSql database providing a service which is comparable to DynamoDB in functionality. We conduct this experiment to better understand the available options for having a scalable key-value store. We use both m1.medium and cc2.8xlarge instances to run ZHT. On 96 nodes scale with 2cc.8xlarge instance type, ZHT offers 1215.0 K ops/s while DynamoDB failed the test since it saturated the capacity. The maximum measured throughput of DynamoDB was 11.5K ops/s which is found at 64 cc2.8xlarge instance scale. For a fair comparison, both DynamoDB and ZHT have 8 clients per node.

Figure 14 shows that the throughput of ZHT on m1.medium and cc2.8xlarge instances are respectively 59x and 559x higher than DynamoDB on 1 instance scale. On the 96 instance scale they are 20x and 134x higher than the DynamoDB.

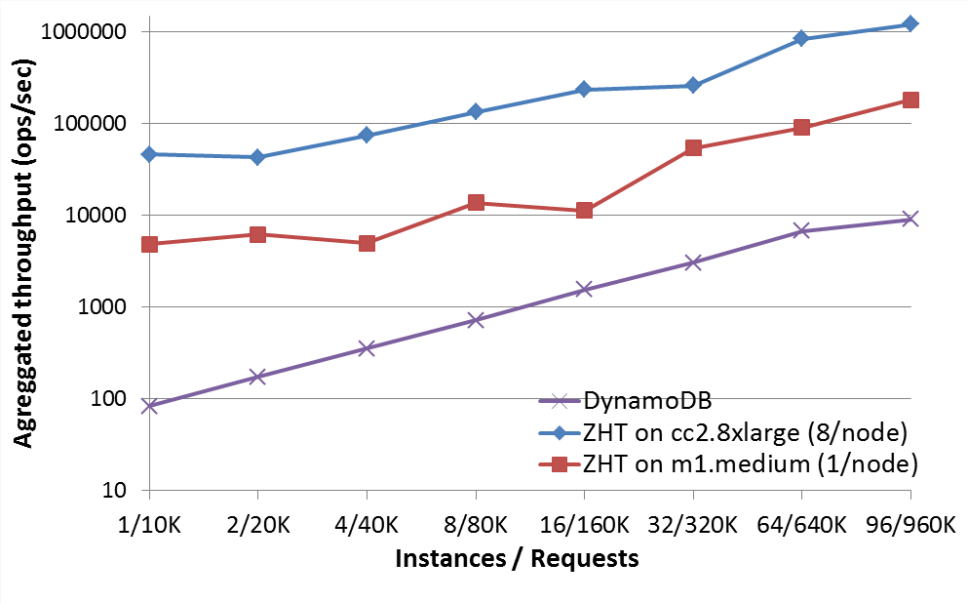

Figure 14. Throughput comparison of DynamoDB with ZHT running on m1.medium and cc2.8xlarge instances on different scales.

i. Workflow Application Performance

In this section we analyze the performance of a complex scientific computing application on the Amazon EC2 cloud. The application investigated is Power Locational Marginal Price Simulation (LMPS), and it is coordinated and run through the Swift parallel programming system [32]. Optimal power flow studies are crucial in understanding the flow and price patterns in electricity under different demand and network conditions. A big computational challenge arising in power grid analysis is that simulations need to be run at high time resolutions in order to capture effect occurring at multiple time scales. For instance, power flows tend to be more constrained at certain times of the day and of the year, and these need to be identified.

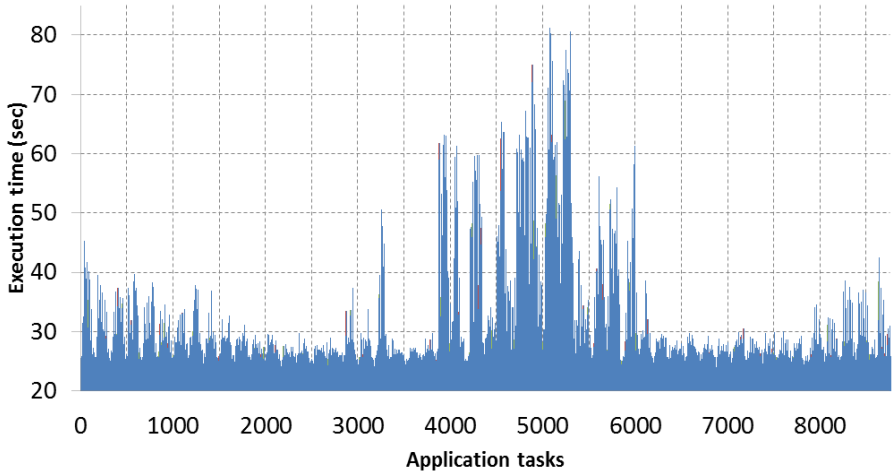

Figure 15. The LMPS application tasks time distributions.

The power flow simulation application under study analyzes historical conditions in the Illinois grid to simulate instant power prices on an hourly basis. The application runs linear programming solvers invoked via an AMPL (A Mathematical Programming Language) representation and collects flow, generation, and price data with attached geographical coordinates [42]. A typical application consists of running the model in 8760 independent executions corresponding to each hour of the year. Each application task execution spans in the range between 25 and 80 seconds as shown in the application tasks time distribution graph in **Figure 15**. A snapshot of one such result prices plotted over the map of Illinois is shown in **Figure 16**. The prices are in US dollars per Megawatt-hour shown as interpolated contour plots across the areas connected by transmission lines and generation stations shown as lines and circles respectively. A series of such plots could be post processed to give an animated visualization for further analysis in trends etc.

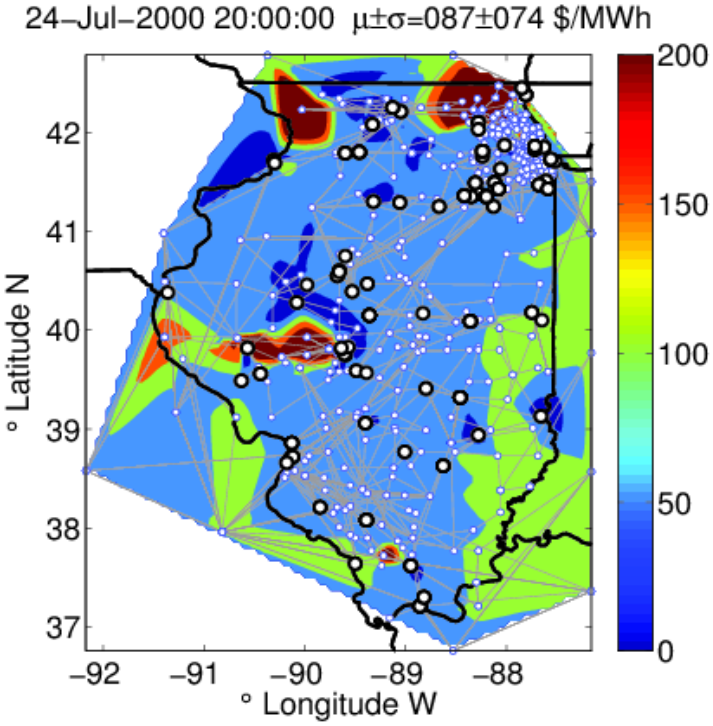

Figure 16. A contour plot snapshot of the power prices in $/MWh across the state of Illinois for an instance in July 2000

The execution of the application was performed on an increasing number of m1.large instances (see **Figure 17**).

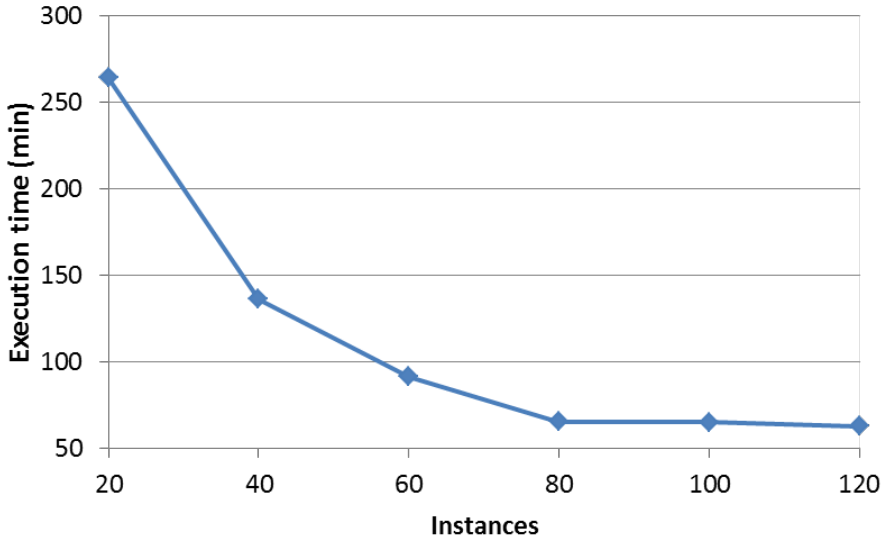

Figure 17. The runtime of LMPS on m1.large instances in different scales.

For data storage, we use S3. Given that the application scales well to 80 instances, but not beyond that. The performance saturation is a salient point that comes out of **Figure 17**. With S3 object store being remote, at 100 VMs it takes long enough to fetch the data that it is dominating execution time. More scalable distributed storage subsystem should be investigated that is geared towards scientific computing, such as PVFS, Lustre, or GPFS.

1.1.1 Performance Comparison of EC2 vs. FermiCloud

In this section we compare the performance of the EC2 as a public cloud with FermiCloud as a private cloud on HPL benchmark which is a real HPC application. Before comparing the performance of Amazon on real Applications, we need to compare the raw performance of the two resources.

a. Raw performance comparison

Before comparing the performance of the two infrastructures on real applications like HPL, we need to compare their raw performance on the essential metrics in order to find the root causes of their performance differences. The most effective factors on HPL performance are compute power, and Network latency and bandwidth. We need to compare these factors on the instances with similar functionalities.

On both of the Clouds, we chose the instances that can achieve the highest performance on HPL applications. On EC2, we use c3.8xlarge instances that are enabled with Intel Xeon E5-2680 v2 (Ivy Bridge) Processors and a 10 Gigabits network adapter with SRIOV technology. On FermiCloud, each server machine is enabled with 2 quad core 2.66 GHz Intel processors, and 8 port RAID Controller. On FermiCloud machines are backed by (16 Gigabits effective) Infiniband network adapters.

The CPU efficiency is defined as the performance of the VM running HPL on a single VM with no network connectivity, divided by the theoretical peak performance of the CPU. **Figure 18** compares the raw performance of the Amazon EC2 with FermiCloud on CPU and network performance. The results show that the virtualization overhead on FermiCloud instances are slightly lower than the EC2 instances.

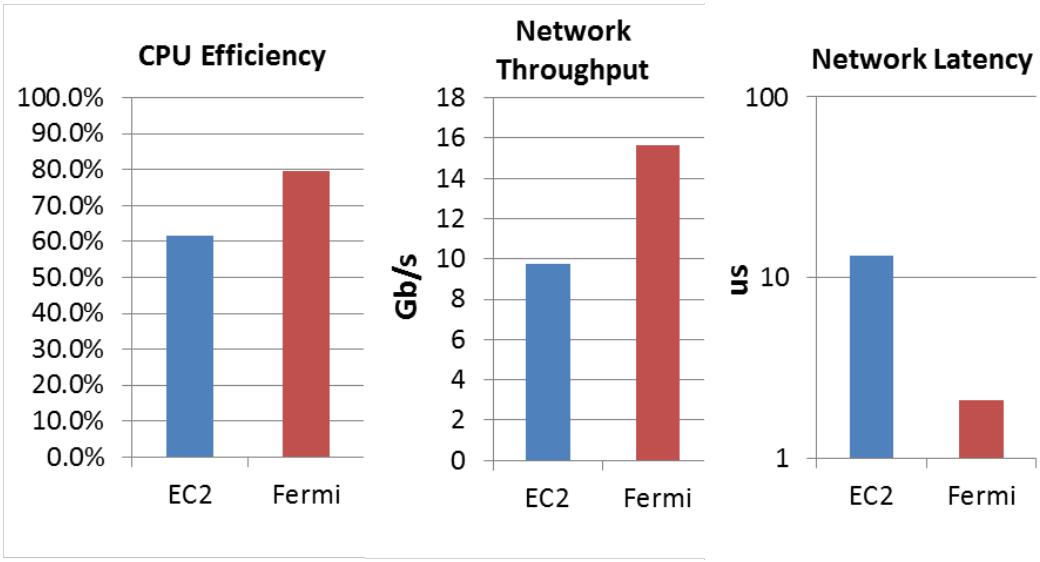

Figure 18. Raw performance comparison overview of EC2 vs. FermiCloud

The significant difference of the two infrastructures is on the network adapters. The FermiCloud instances are enabled with InfiniBand network adapters and are able to provide higher performance compared to the EC2 instances that have 10 Gigabit network cards. The efficiency of both of the systems on network throughput is high. The network throughput efficiency is defined as the VM network performance divided by the theoretical peak of the device. FermiCloud and EC2 network adapters respectively achieve 97.9% and 97.4% efficiency. We used MPIbench to calculate the network latency. There is a 6x difference between the network latency of the two clouds. The latency of the FermiCloud instance is 2.2 us as compared to the latency of EC2 instance which is 13 us. Another important factor is the latency variance. The latency variance on both systems is within 20% which is stable. HPL application uses MPI for communication among the nodes. The network latency can decrease the performance of the application by affecting the MPI performance.

b. HPL performance comparison

In this section we evaluate the performance of HPL application on both on a virtual cluster on both FermiCloud and EC2. The main difference on the two infrastructures is on their virtualization layer and the network performance. FermiCloud uses KVM and is enabled with InfiniBand network adapters. EC2 uses its own type of virtualization which is based on Xen hypervisor and has 10 Gigabit network adapters.

The best way to measure the efficiency of a virtual cluster on a cloud environment is defining it as the performance of the VM which include the virtualization overhead divided by the host performance that doesn't include virtualization overhead. We can measure the efficiency as defined for FermiCloud since we have access to the host machines. But that is not possible for EC2 since we do not have access to the physical host machines. Therefore we compare the scalability efficiency of the two clouds which is defined as the overhead of the application performance as we scale up the number of cloud instances.

Figure **19** compares the efficiency of EC2 and FermiCloud running HPL application on a virtual cluster. Due to budget limitations we run the experiment up to 32 instances scale.

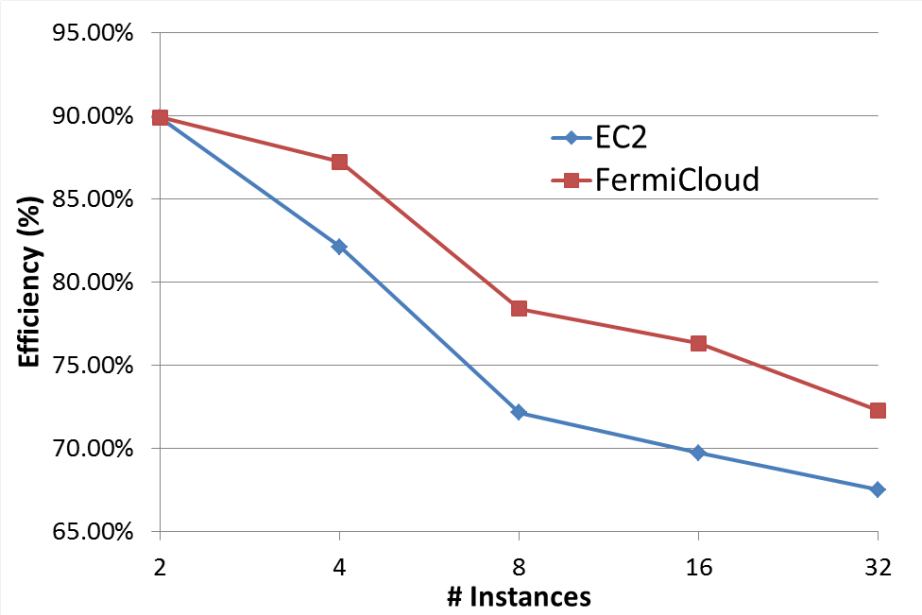

Figure 19. Efficiency comparison of EC2 and FermiCloud running HPL application on a virtual cluster.

The results show that the efficiency is dependent on the network latency. On the 2 instances scale, both clouds show good efficiency. They only lose 10% efficiency that is due to the MPI communications latency added between the instances. Since both of the clouds have relatively powerful network adapters, the communication overhead is still not a bottleneck on 2 instances scale. As the number of instances increase, the applications processes make more MPI calls to each other and start saturating the network bandwidth. Having InfiniBand network, the FermiCloud loses less efficiency than the EC2. The efficiency of EC2 drops to 82% and the efficiency of the FermiCloud drops to 87%. The only major difference between the instances of private and public cloud is on their network latency. As a result, we can see that they provide similar

efficiency with the private cloud instance being roughly about 5-8% more efficient on different scales.

2.3 Cost Analysis

In this section we analyze the cost of the Amazon EC2 cloud from different aspects. We analyze the cost of in-stances for compute intensive applications as well as for data intensive applications. Our analysis provides suggestions to different cloud users to find the instance type that fits best for certain application with specific requirements. Next section compares the instances based on their memory capacity and performance.

2.3.1 Memory cost

This section compares the cost of the memory on Amazon EC2 instances. Figure 20 compares the cost of instances based on their memory capacity and bandwidth. The GB/Dollar metric on the left hand side shows the capacity cost effectiveness of the instances. The most cost effective instances for memory capacity are the high memory (m2.2xlarge & m2.4xlarge) instances. But looking at the cost of the memory bandwidth, we can observe that these instances do not have the best memory bandwidth efficiency. The most cost effective instances based on the memory bandwidth efficiency are the m1.small and m1.medium instances.

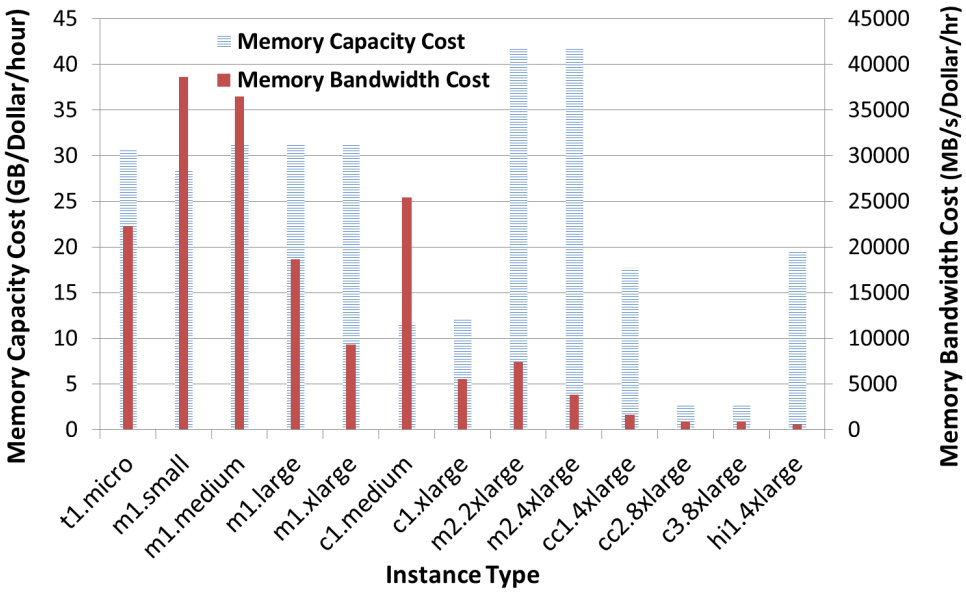

Figure 20. Memory capacity and memory bandwidth cost.

2.3.2 CPU cost

In this section we analyze the cost-effectiveness of in-stances based on the performance of the instances while running compute intensive applications. The metric for our analysis is GFLOPS/Dollar.

Figure 21 compares the ideal performance cost of the in-stances based on Amazon claims with their actual performance while running HPL benchmark. The results show that the most cost-effective instance is c3.8xlarge.

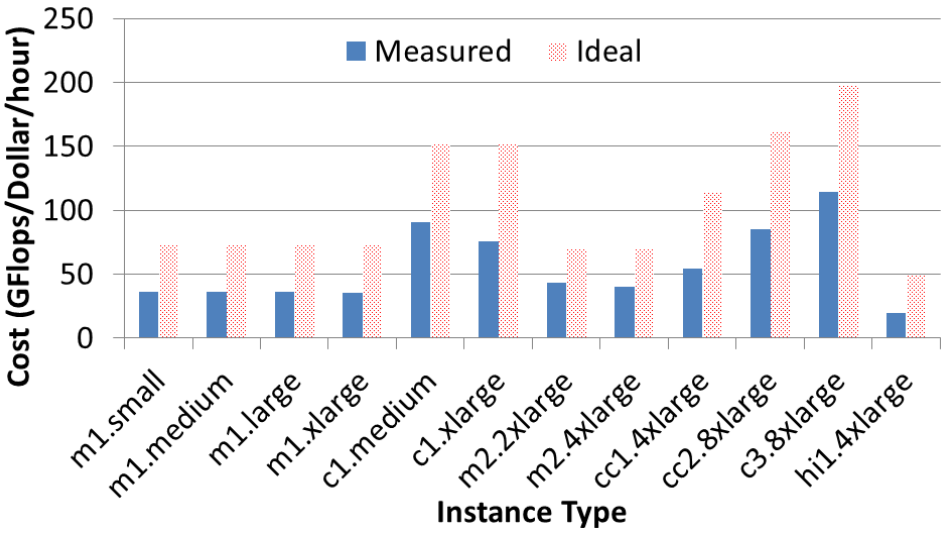

Figure 21. CPU performance cost of instances

2.3.3 Cluster cost

We analyze the cost of the virtual clusters set up by m1.medium and cc1.4xlarge instances in different sizes. Figure 22 compares the cost of the virtual clusters based on their compute performance.

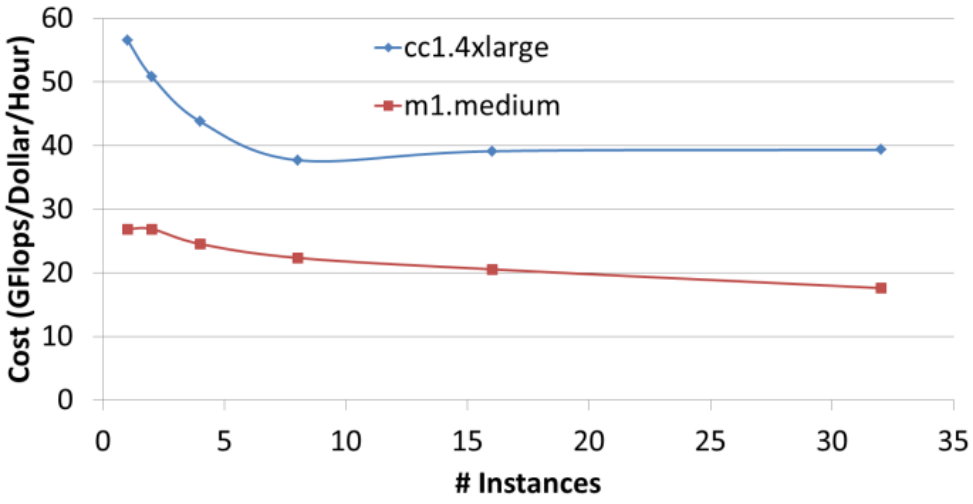

Figure 22. Cost of virtual cluster of m1.medium and cc1.4xlarge.

2.3.4 DynamoDB cost

Finally in this section we evaluate the cost of DynamoDB. In order to better understand the value of offered service, we compare the cost with the cost of running ZHT on EC2 on different instance types.

Figure 23 shows the hourly cost of 1000 ops/s capacity offered by DynamoDB compared to the equal capacity provided by ZHT from the user point of view.

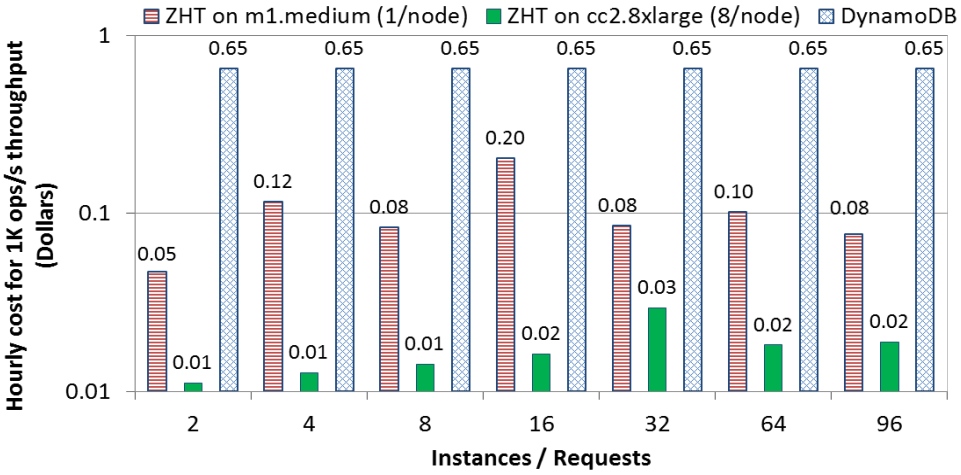

Figure 23. Cost Comparison of DynamoDB with ZHT

We are comparing the two different scenarios of cost of using a free application on rented EC2 instances versus getting the service from DynamoDB. In case of DynamoDB, since the users pays for the capacity that they get, the number of instances doesn't affect the cost. That's why the cost of DynamoDB is always constant. For ZHT, the system efficiency and performance varies on different scales hence the variation in costs for ZHT at different scales. Since the cc2.8xlarge instances provide much better performance per money spent, the cost per operation is as good as 65X lower than DynamoDB. However, the better costs come at the complexity of managing a virtual cluster of machines to

operate ZHT. It is likely that for low loads including sporadic requirements for DynamoDB, it makes financial sense to run on Amazon AWS services, but for higher performance requirements it is much more beneficial to simply operate a dedicated ZHT system over EC2 resources.

2.3.5 Performance and cost summary

This section summarizes the performance and the cost efficiency of Amazon EC2 and other services of AWS. Table 1 shows the performance overview of the different instance types on EC2. The performance results of the instances mostly match with the prediction based on the claims of Amazon. There have been anomalies in some of the specific instance types. Instances like m1.xlarge have average performance while m1.medium instance has shown a performance that was higher than expected.

Table 1: Performance summary of EC2 Instances

	CPU bw	Mem. bw	Net. bw	Disk I/O
m1.small	Low	Low	Low	Low
m1.med	Low	Avg	Avg	Low
m1.lrg	Avg	Avg	Avg	Avg
m1.xlrg	Avg	Avg	Avg	Avg
c1.med	Avg	Avg	Avg	Low
c1.xlrg	Avg	High	Avg	Avg
m2.2xlrg	High	High	Avg	Avg
cc1.4xlrg	High	High	High	Avg
cc2.8xlrg	High	High	High	Avg
c3.8xlrg	High	High	High	High
hi1.lrg	High	Avg	High	High

Table 2 summarizes the cost-efficiency of instance types of EC2. The compute optimized instances show better cost efficiency. Finally table 3 summarizes the performance of S3 and DynamoDB.

Table 2: Cost-efficiency summary of EC2 Instances

	CPU bw	Mem. Cap.	Mem. bw	Net. bw
m1.small	Avg	Avg	High	High
m1.med	Avg	Avg	High	High
m1.lrg	Avg	Avg	Avg	Avg
m1.xlrg	Avg	Avg	Low	Low
c1.med	High	Low	High	Low
c1.xlrg	High	Low	Low	Low
m2.2xlrg	Avg	High	Low	Low
cc1.4xlrg	Avg	Avg	Low	Low
cc2.8xlrg	High	Avg	Low	Avg
c3.8xlrg	High	Avg	Low	Avg
hi1.lrg	Low	Low	Low	Low

Table 3: Performance and Cost-efficiency summary of AWS Services

	Scalability	Cost-efficiency	Data Granularity
S3	High	High	Large data
DynamoDB	High	Low	Small data

2.4 Summary

In this chapter, we present a comprehensive, quantitative study to evaluate the performance of the Amazon EC2 for the goal of running scientific applications. We first evaluate the performance of various instance types by running micro benchmarks on memory, compute, network and storage. In most of the cases, the actual performance of the instances is lower than the expected performance that is claimed by Amazon. The network bandwidth is relatively stable. The network latency is higher and less stable than what is available on the supercomputers. Next, based on the performance of instances on micro-benchmarks, we run scientific applications on certain instances. We finally

compare the performance of EC2 as a commonly used public cloud with FermiCloud, which is a higher-end private cloud that is tailored for scientific for scientific computing.

We compare the raw performance as well as the performance of the real applications on virtual clusters with multiple HPC instances. The performance and efficiency of the two infrastructures is quite similar. Their only difference that affects their efficiency on scientific applications is the network bandwidth and latency which is higher on FermiCloud. FermiCloud achieves higher performance and efficiency due to having InfiniBand network cards. We can conclude that there is need for cloud infrastructures with more powerful network capacity that are more suitable to run scientific applications.

We evaluated the I/O performance of Amazon instances and storage services like EBS and S3. The I/O performance of the instances is lower than performance of dedicated resources. The only instance type that shows promising results is the high-IO instances that have SSD drives on them. The performance of different parallel file systems is lower than performance of them on dedicated clusters. The read and write throughput of S3 is lower than a local storage. Therefore it could not be a suitable option for scientific applications. However it shows promising scalability that makes it a better option on larger scale computations. The performance of PVFS2 over EC2 is convincible for using in scientific applications that require a parallel file system.

Amazon EC2 provides powerful instances that are capable of running HPC applications. However, the performance a major portion of the HPC applications are heavily dependent on network bandwidth, and the network performance of Amazon EC2 instances cannot keep up with their compute performance while running HPC

applications and become a major bottleneck. Moreover, having the TCP network protocol as the main network protocol, all of the MPI calls on HPC applications are made on top of TCP protocol. That would add a significant overhead to the network performance. Although the new HPC instances have higher network bandwidth, they are still not on par with the non-virtualized HPC systems with high-end network topologies. The cloud instances have shown to be performing very well, while running embarrassingly parallel programs that have minimal interaction between the nodes [19]. The performance of embarrassingly parallel application with minimal communication on Amazon EC2 instances is reported to be comparable with non-virtualized environments [37][39]. Armed with both detailed benchmarks to gauge expected performance and a detailed price/cost analysis, we expect that this chapter will be a recipe cookbook for scientists to help them decide between dedicated resources, cloud resources, or some combination, for their particular scientific computing workload.

CHAPTER 3

ACHIEVING EFFICIENT DISTRIBUTED SCHEDULING WITH MESSAGE QUEUES IN THE CLOUD FOR MANY-TASK COMPUTING AND HIGH-PERFORMANCE COMPUTING

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. In this chapter, we design a compact, light-weight, scalable, and distributed task execution framework (CloudKon) that builds upon cloud computing building blocks (Amazon EC2, SQS, and DynamoDB).

3.1 Background and Motivation

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.

The architecture of commonly used schedulers have a centralized manager (e.g. Slurm [4], Condor [6]), with a central server that is responsible for resource management and the job allocation. This architecture seems to be working fine with today's

infrastructure scales. However, this trend is less likely to continue like this. The centralized architecture cannot scale well with the next generation distributed systems. Also, having a central controller could become a single point of failure. To solve this problem decentralized architectures have been proposed. Distributed schedulers are normally implemented in either hierarchical [20] or fully distributed architectures [19] to address the scalability issue. Those solutions can solve the problem of the single point of failure. But more problems arise in load balancing, resource utilization and the information synchronization.

The idea of using cloud computing for scientific applications have been explored in other research works. However, most of these works have approached the cloud as yet another distributed resource with similar characteristics with traditional resources [21][53][15]. Sharing its physical resources and using virtualization makes public clouds totally different than the traditional HPC systems. The uniqueness of our work is in proposing a new approach. We offer to utilize public cloud's native integrated resources for more efficient performance. Moreover, using cloud services enables programmers to create fairly complicated systems with a shorter code base and in a shorter period of time. ***In this chapter, 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.***

Today's data analytics are moving towards interactive shorter jobs with higher throughput and shorter latency [20][69]. 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 [71].

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 chapter are as follows. Section 3.2 discusses about the design and implementation details of CloudKon. Section 3.3 evaluates the performance of the CloudKon in different aspects using different metrics. Finally section 3.4 discusses about the limitations of the current work, and covers the future directions of this work.

3.2 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 distribute 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.

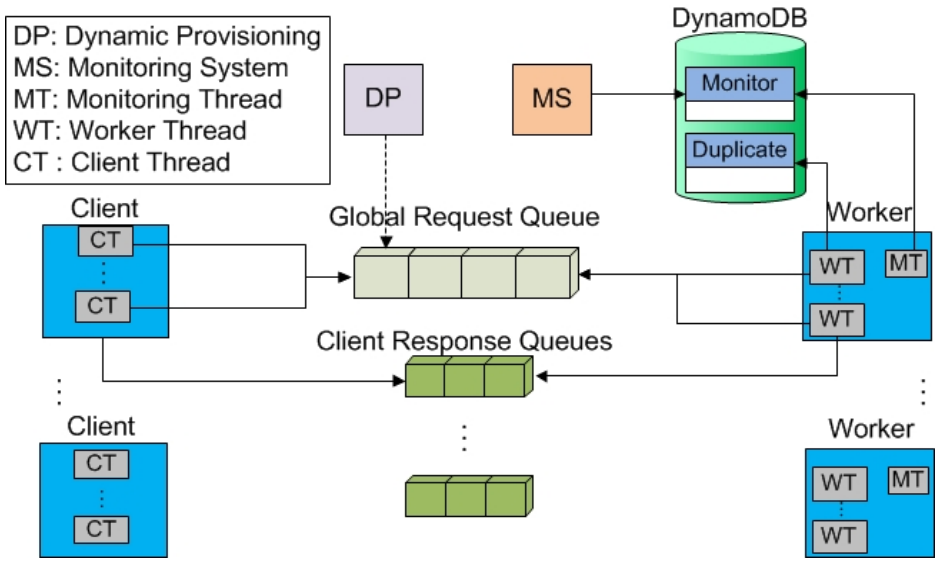

Figure 24. CloudKon architecture overview

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.

3.2.1 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.

a. MTC task management

Figure 24 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 [123] 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.

b. HPC task management

Figure 25 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.

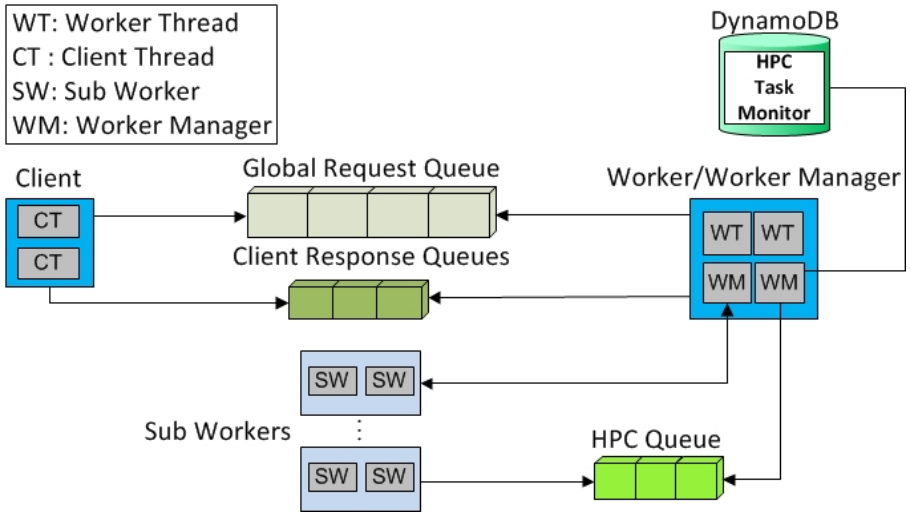

Figure 25. CloudKon-HPC architecture overview

3.2.2 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.

3.2.3 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 [77]. 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.

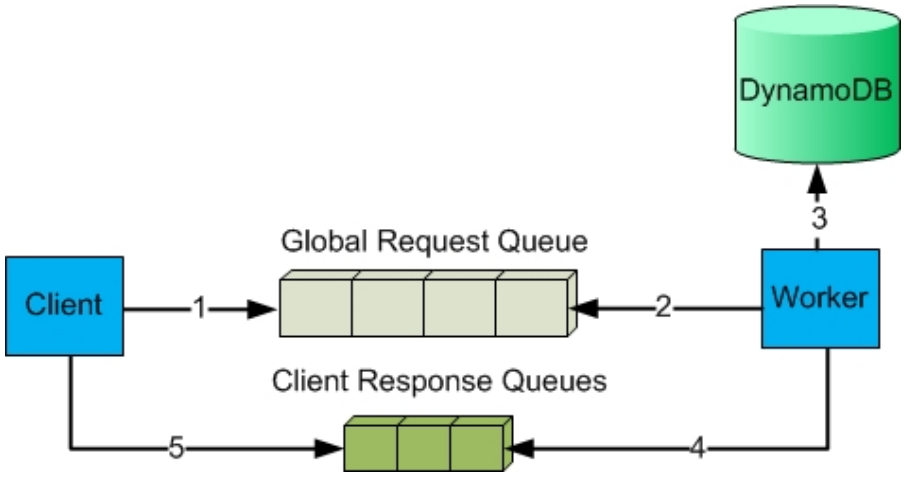

Figure 26. Communication Cost

3.2.3 Communication costs

The network latency between the instances in the public Cloud is relatively high compared to HPC systems[70][71]. In order to achieve reasonable throughput and latency we need to minimize the communication overhead between the different components of the system. Figure 26 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.

3.2.4 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 [56].

3.2.5 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, while 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.

3.3 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.

3.3.1 CloudKon vs. 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 system. 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 [106]. 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.

3.3.2 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 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 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.

3.3.3 Throughput

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 27 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 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. 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 significantly lower than that of HPC systems, where MATRIX has successfully been run at 1024-node scales.

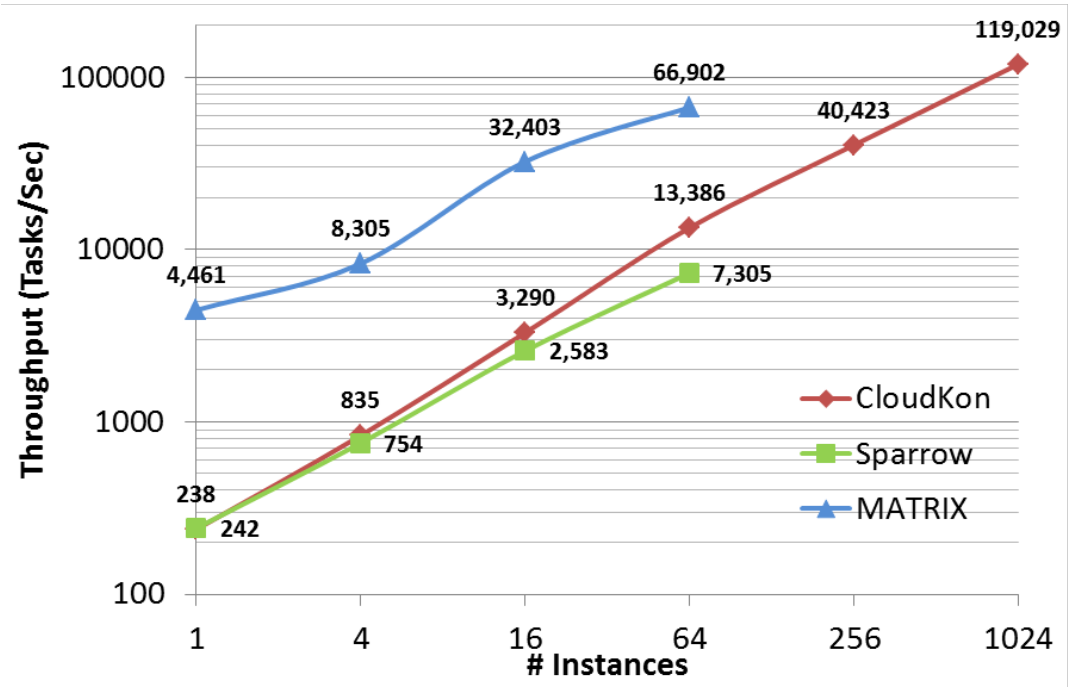

Figure 27. Throughput of CloudKon, Sparrow and MATRIX (MTC tasks)

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.

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 28 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.

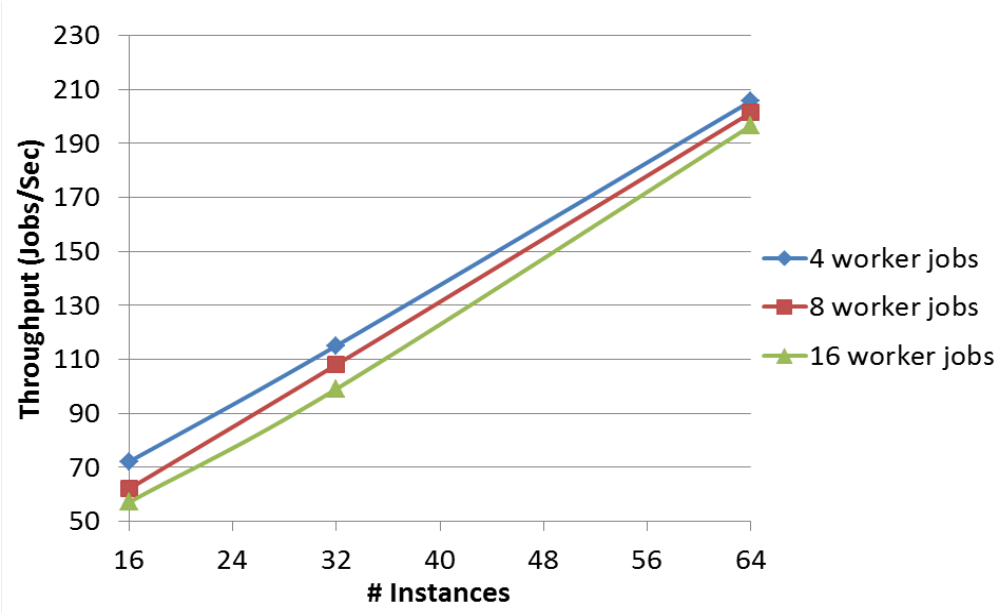

Figure 28. Throughput of CloudKon (HPC tasks)

3.3.4 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 29 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.

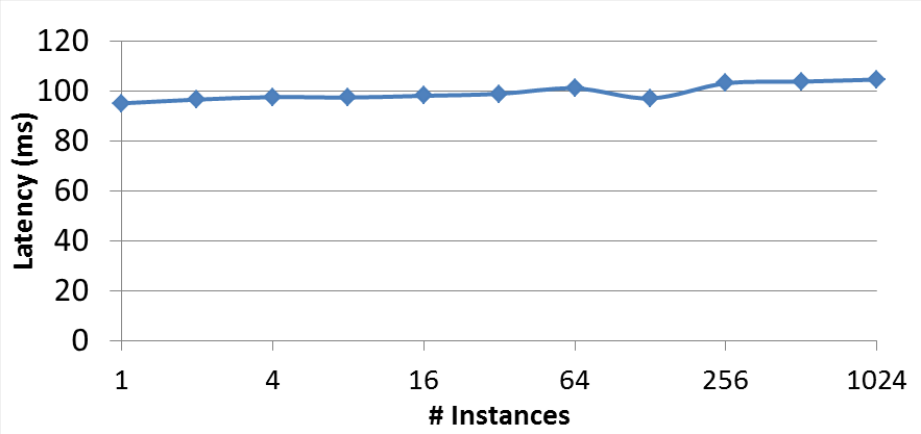

Figure 29. Latency of CloudKon sleep 0 ms tasks

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 29 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 30, Figure 31, and Figure 32 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.

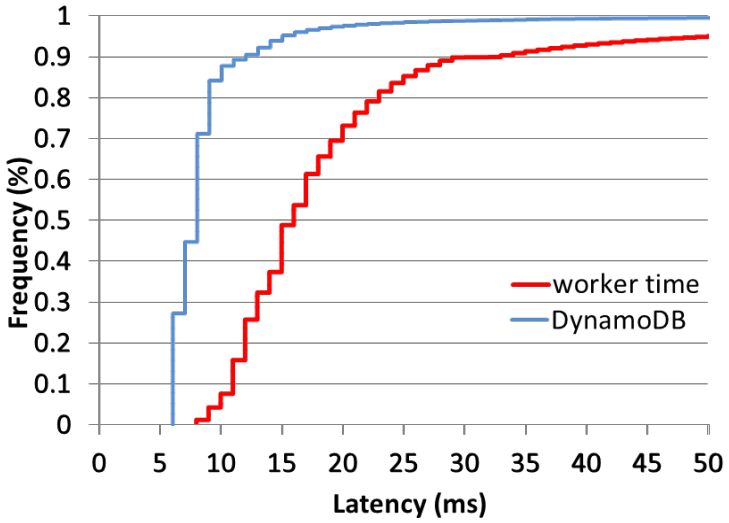

Figure 30. Cumulative Distribution of the latency on the task execution step

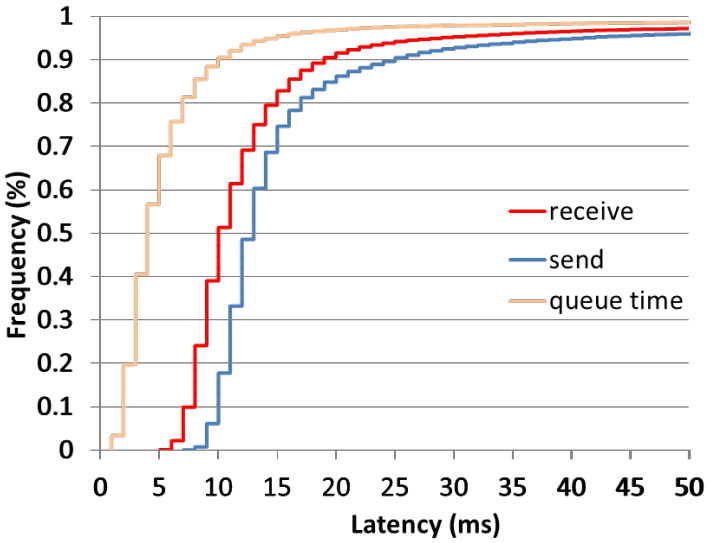

Figure 31. Cumulative Distribution of the latency on the task submit step

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 much longer than the time it spends on request-queue. The reason for that 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.

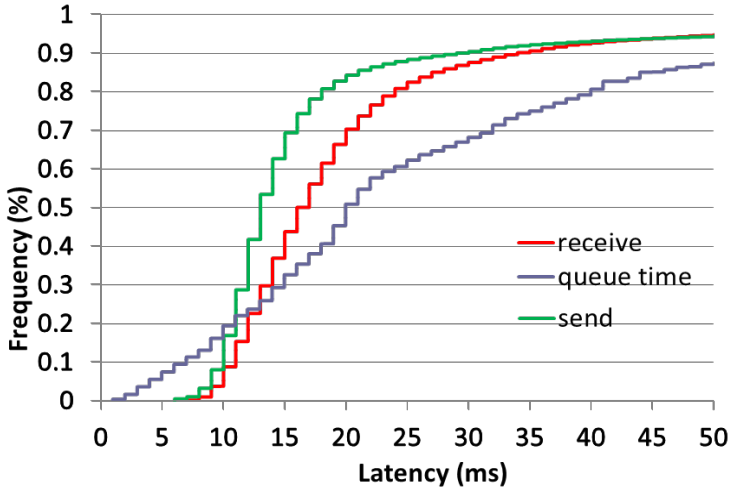

Figure 32. Cumulative Distribution of the latency on the result delivery step

3.3.5 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.

Homogeneous Workloads

In this section we compare the efficiency of CloudKon with Sparrow and MATRIX on sub second tasks. Figure 33 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 128 ms 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.

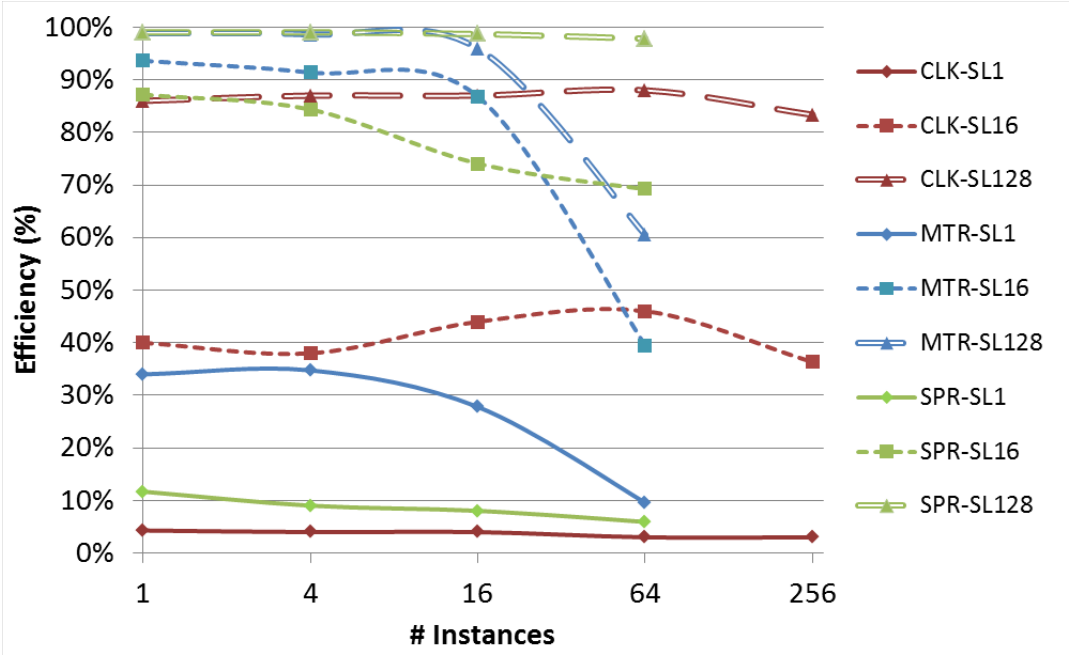

Figure 33. Efficiency of CloudKon, Sparrow and MATRIX running homogenous workloads of different task lengths (1, 16, 128ms tasks)

Heterogeneous Workloads

In order to measure efficiency, we investigated the largest available trace of real MTC workloads [72], 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 34 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.

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.

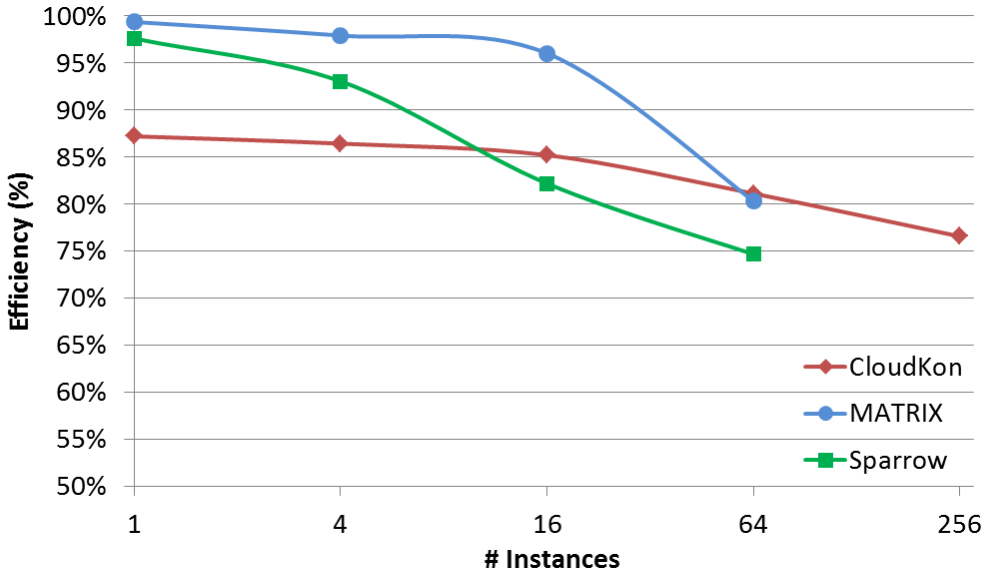

Figure 34. Efficiency of the systems running heterogeneous workloads.

3.3.6 The overhead of consistency

In this section we evaluate effect of tasks execution consistency on CloudKon. Figure 35 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 chapter.

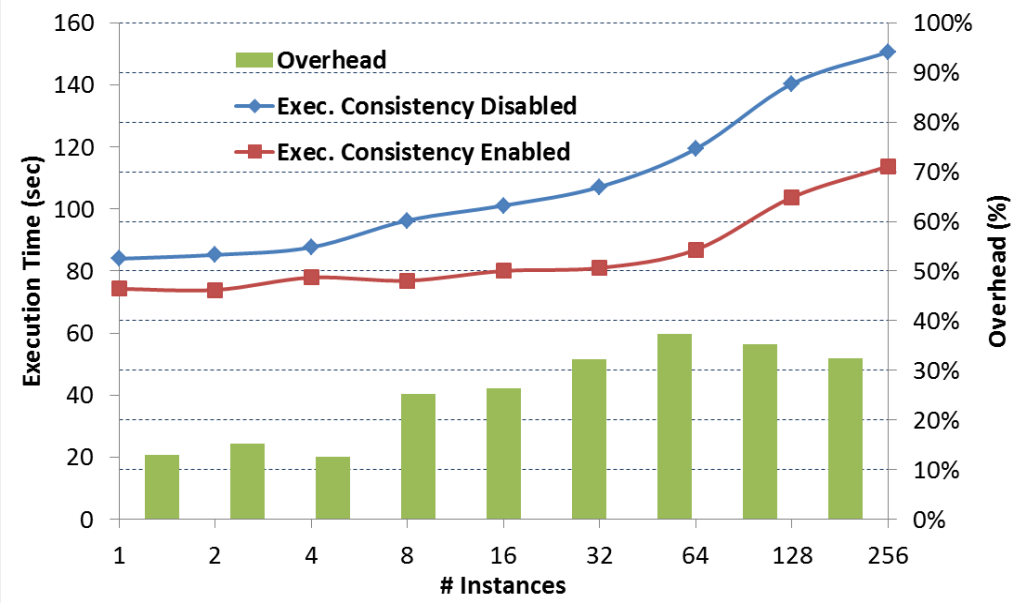

Figure 35. The overhead of task execution consistency on CloudKon

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.

3.4 Summary

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 [22], 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 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.

CHAPTER 4

FABRIQ: LEVERAGING DISTRIBUTED HASH TABLES TOWARDS DISTRIBUTED PUBLISH-SUBSCRIBE MESSAGE QUEUES

CloudKon was able to achieve good performance scalability compared to other state of the art works. However it has its own limitations. CloudKon uses SQS as its building block. Therefore, it is not possible to use CloudKon in other distributed resources. That has driven us to design and implement a fully distributed message queuing service. That enables us to run CloudKon in other environments including private clouds, other public clouds, and even HPC resources. Moreover, we can integrate this queue within the CloudKon and achieve significant improvement over latency and efficiency. In this chapter, we propose Fabriq, a distributed message queue that runs on top of a Distributed Hash Table. The design goal of Fabriq is to achieve lower latency and higher efficiency while being able to handle large scales.

4.1 Background and Motivation

With the growth of the data at the current rate, it is unlikely for the traditional data processing systems that usually tend to structure the data, to be able to handle the requirement of Big Data processing. There is a need to reinvent the wheel instead of using the traditional systems. Traditional data processing middleware and tools such as SQL databases and file system are being replaced by No-SQL data-stores and key-value storage systems in order to be able to handle the data processing at the current scale.

Another key tool that is getting more attention from the industry is distributed queuing service [104][108][109].

A Distributed Message Queue could play an important role as a middleware for today's Big Data requirements. A message queue could be a key part of a loosely coupled distributed application. Over the past few years, distributed queuing services have been used in both industrial and scientific applications and frameworks [75][46][2][88][92]. SQS is a distributed queue service by Amazon AWS, which is being leveraged by various commercial applications. Some systems have used SQS as a buffer for their server to handle massive number of requests. Other applications have used SQS in monitoring, workflow applications, big data analytics, log processing and many other distributed systems scenarios [105][75][89].

The large scale log generation and processing is another example that has become a major challenge on companies that have to deal with the big data. Many companies have chosen to use distributed queue services to address this challenge. Companies like LinkedIn, Facebook [79], Cloudera [77] and Yahoo have developed similar queuing solutions to handle gathering and processing of terabytes of log data on their servers [78]. For example LinkedIn's Kafka [76] feeds hundreds of gigabytes of data into Hadoop [91] clusters and other servers every day.

Distributed Queues can play an important role in Many Task Computing (MTC) [53] and High Performance Computing (HPC). Modern Distributed Queues can handle data movement on HPC and MTC workloads in larger scales without adding significant overhead to the execution [80].

CloudKon is a Distributed Job Scheduling system that is optimized to handle MTC and HPC jobs. It leverages SQS as a task delivery fabric that could be accessed simultaneously and achieve load balancing at scale [46]. CloudKon has proved to outperform other state-of-the-art schedulers like Sparrow [14] by more than 2X in throughput. One of the main motivations of this work is to provide a DMQ that can replace SQS in future versions of the CloudKon. There are a few limitations with SQS including having duplicate messages, and getting the system tied to AWS cloud environment. CloudKon uses DynamoDB [75] to filter out the duplicate messages.

There are various commercial and open sourced queuing services available [81][82][83][84]. However, they have many limitations. Traditional queue services usually have centralized architecture and cannot scale well to handle today's big data requirements. Providing features such as transactional support or consumption acknowledgement makes it almost impossible for these queues to achieve low latency. Another important feature is persistence. Many of the currently available options are in memory queues and cannot guarantee persistence. There are only a few DMQs that can scale to today's data analytics requirement. Kafka is one of those that provides large scale message delivery with high throughput. However, as it is shown in Figure 31 and Figure 32, Kafka has a long message delivery latency range. Moreover, as we have shown in Figure 29, Kafka cannot provide a good load balance among its nodes. That could cause Kafka to perform inefficiently in larger scales.

Today's data analytics applications have moved from coarse granular tasks to fine granular tasks which are shorter in duration and much more in number [14]. Such

applications cannot tolerate a data delivery middleware with an overhead in the order of seconds. It is necessary for a DMQ to be as efficient as possible without adding substantial overhead to the workflow.

We propose Fast, Balanced and Reliable Distributed Message Queue (Fabriq), a persistent reliable message queue that aims to achieve high throughput and low latency while keeping the near perfect load balance even on large scales. Fabriq uses ZHT as its building block. ZHT is a persistent distributed hash table that allows low latency operations and is able to scale up to more than 8k-nodes [23]. Fabriq leverages ZHT components to support persistence, consistency and reliable messaging. Message delivery guarantee is a necessity for a DMQ. This requirement becomes a challenge for the systems that aim to support large scale delivery. A common practice is to keep multiple copies of the message on multiple servers of a DMQ. Once the message gets delivered by a server, it will asynchronously inform other servers to remove their local copies. However, since the informing process is asynchronous, there is a change of having a message delivered to multiple clients before getting removed from the servers. Hence, the systems with such procedure can generate duplicate messages.

The fact that Fabriq provides low latency makes it a good fit for HPC and MTC workloads that are sensitive to latency and require high performance. Also, unlike the other compared systems (Figure 31 and Figure 32), Fabriq provides a very stable delivery in terms of latency variance. Providing a stable latency could be substantial for MTC applications, as well as towards having predictable performance. Finally, Fabriq supports

dynamic scale up/down during the operation. In summary, the contributions of Fabriq are:

- It uses ZHT as its building block to implement a scalable DMQ.
- Leveraging ZHT components, it supports persistence, consistency, reliable messaging and dynamic scalability.
- It guarantees the at least once delivery of the messages.
- It achieves a near perfect load balance among its servers.
- It provides high throughput and low latency outperforming Kafka and SQS. It also provides a shorter latency variance than the other two systems.
- It could be used on HPC environments that do not support Java (e.g. Blue Gene L/P supercomputers).

The rest of this chapter is organized as follows. Section 4.2 discusses the Fabriq's architecture. We first briefly go over the architecture of ZHT and explain how Fabriq leverages ZHT to provide an efficient and scalable DMQ. Later on section 4.3, we analyze the communication costs on Fabriq. Section 4.4 evaluates the performance of the Fabriq in different metrics. Finally, section 4.5 summarizes this chapter and discusses about the future work.

4.2 Fabriq Architecture and Design Principles

This section discusses the Fabriq design goals and demonstrates its architecture. As we discussed in the previous section, many of the available alternative solutions do not guarantee persistence and reliability. There are many ways to implement a distributed queue. A distributed queue should be able to guarantee message delivery. It should also

be reliable. Finally, a distributed queue has to be highly scalable. Most of the straight forward design options include centralized manager component that limit the scalability. Depending on the architecture, a Distributed Hash Table (DHT) could achieve high scalability as well as maintaining other benefits. Fabriq uses a DHT as its building block of our queuing services. The simple put and get methods of a DHT could be similar to the push and pop methods on a DMQ. We chose to use ZHT which is a low overhead and low latency DHT, and has a constant routing time. It also supports persistence. Before discussing the design details of Fabriq, we briefly review the architecture and the key features of ZHT

4.2.1 ZHT overview

ZHT has a simple API with 4 major methods: 1. *insert*(key, value); 2. *lookup*(key); 3. *remove*(key), and 4. *append*(key,value). The key in ZHT is a simple ASCII character string, and the value can be a complex object. A key look up in ZHT can take from 0 (if the key exists in the local server) to 2 network communications. This helps providing the fastest possible look up in a scalable DHT. The following sections discuss main features of ZHT.

a) Network Communication

ZHT supports both TCP and UDP protocols. In order to optimize the communication speed, the TCP connections will be cached by a LRU cache. That will make TCP connections almost as fast as UDP. In Fabriq, we rely on the ZHT for the network communications. Having optimized TCP communications enables Fabriq to achieve low latency on its operations.

b) Consistency

ZHT supports replication to provide a reliable service. In order to achieve high throughput ZHT follows a weak consistency model. The first two replications for each dataset are strongly consistent. That means the data will be written to the primary and the secondary replicas. After completion of the write on the secondary replica, the replication to the following replicas happens in an asynchronous fashion.

c) Fault Tolerance

ZHT supports fault tolerance by lazily tagging the servers that are not being responsive. In case of failure, the secondary replica will take the place of the primary replica. Since each ZHT server operates independently from the other servers, the failure of a single server does not affect the system performance. Every change to the in-memory DHT data is also written to the disk. Therefore, in case of system shut down (e.g. reboot, power outage, maintenance, etc.) the entire data could be retrieved from the local disk of the servers.

d) Persistence

ZHT is an in-memory data-structure. In order to provide persistence, ZHT uses its own Non-Volatile Hash Table (NoVoHT). NoVoHT uses a log based persistence mechanism with periodic check-pointing.

e) Dynamic Scalability (membership)

ZHT supports dynamic membership. That means server nodes can join or leave the system any time during the operation. Hence, the system scale can be dynamically changed on ZHT (and also Fabriq) during the operation. Although dynamic scalability of

Fabriq is supported, due to space limitation, we will explore the evaluation of dynamic membership in Fabriq in future work.

4.2.2 Fabriq Design and Architecture

The main design goal of Fabriq is achieving high scalability, efficiency and perfect load balance. Since Fabriq is using ZHT as its building block for saving messages and the communication purposes, and ZHT has proven to be able to scale more than 8k-nodes, we can expect Fabriq to also scale as much as ZHT [23].

Fabriq distributes the queue load of each of the user queue among all of its servers. That means user queues can co-exist on multiple servers. When a single server is down due to any reason such as failure or maintenance, the system can continue serving all of the users with other servers. That enables the system to provide a very high availability and reliability. Figure 36 depicts the message delivery of multiple user queues in Fabriq.

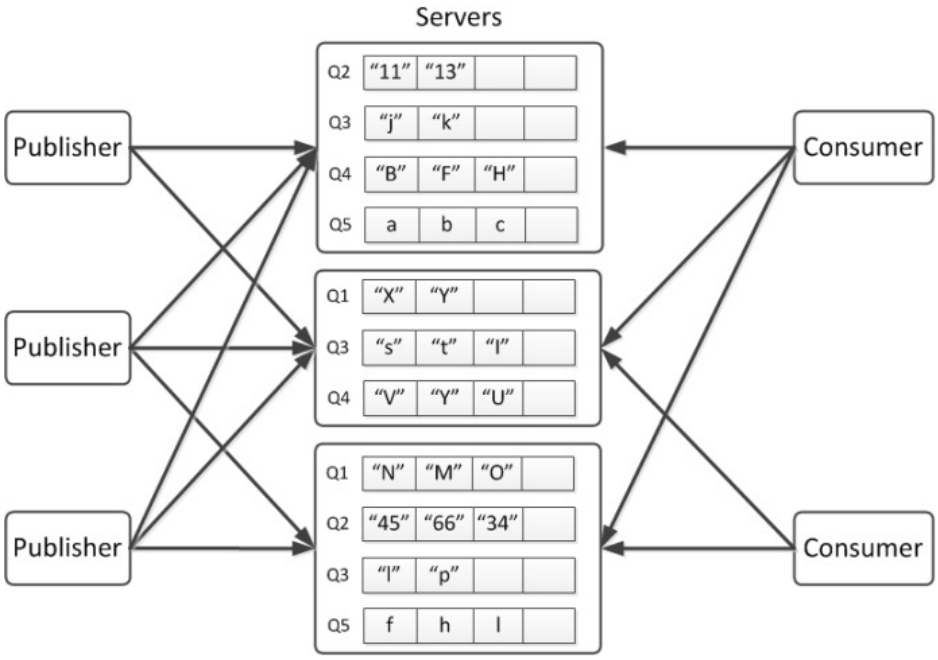

Figure 36. Fabriq servers and clients with many user queues.

Like any other message queue, Fabriq has the simple push and pop functions. In addition to those, Fabriq also supports peek method which is reading the contents of a message without removing it. In order to implement the queue functionalities, we have used ZHT as our system building block and extended the queue functionalities to it. Figure 37 shows the structure of a single Fabriq server. Besides the local NoVoHT hash table, there are two different data structures on each server.

MessageId Queue: it is a local in-memory queue, used to keep the message IDs of a user queue that are saved on the local NoVoHT on this server. The purpose of having this queue is to be able to get messages of a user queue from the local hash table without having the message Ids, and also to distinguish the messages of different user queues from each other. This way, each Fabriq server can independently serve the clients without having the get the message Ids of a user queue from a central server.

Metadata List: each user queue has a unique Metadata list in the whole system which keeps the address of the servers which have messages of this certain queue. The Metadata list only exists in one server. The purpose of having this list is to reduce the chance of accesses to the servers that don't have messages for a user queue.

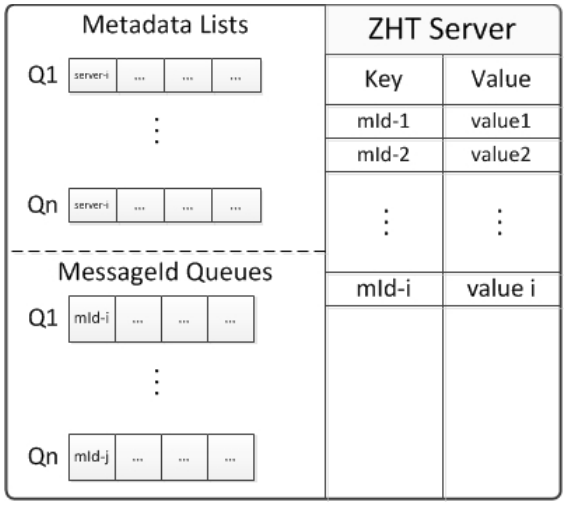

Figure 37. Structure of a Fabriq server.

Next, we discuss about the process of delivering messages by explaining the major methods on Fabriq. Besides the below mentioned methods, Fabriq has peek and deleteQueue.

1) createQueue

This method lets users define their own queue. The method gets a unique name for the queue (assume it is "Qx"), and hashes the name. Based on the hashing value, the client sends a createQueue request to the destination server. Then it will define a unique Metadata List for "Qx". The Metadata List is supposed to keep the address of the servers that keep the messages of "Qx". It will also create a MessageId queue for "Qx" for the future incoming messages to this server. A user queue can have more than one MessageId queue in the whole system, but it has only one Metadata List. The Metadata List of a user queue resides on the server with the same address as the hash value of that user queue name

2) push

Once a user queue has been defined, the client can push messages to it. The method has two inputs: the queue name and the message contents. Fabriq uses Google Protocol Buffer for message serialization and encoding. Therefore, the message contents input supports both string or user defined objects. Once the push method is called, the client first generates a message Id using its IP Address, port, and a counter. The message Id is unique on the whole system. Then, the client hashes the message Id and chooses the destination server based on the hash value. Since the hashing function in Fabriq distributes the signature uniformly among all of the servers, the message could land on any of the collaborating servers. Figure 38 depicts the push procedure on Fabriq. After receiving the push request, the destination server performs one of the following based on the queue name:

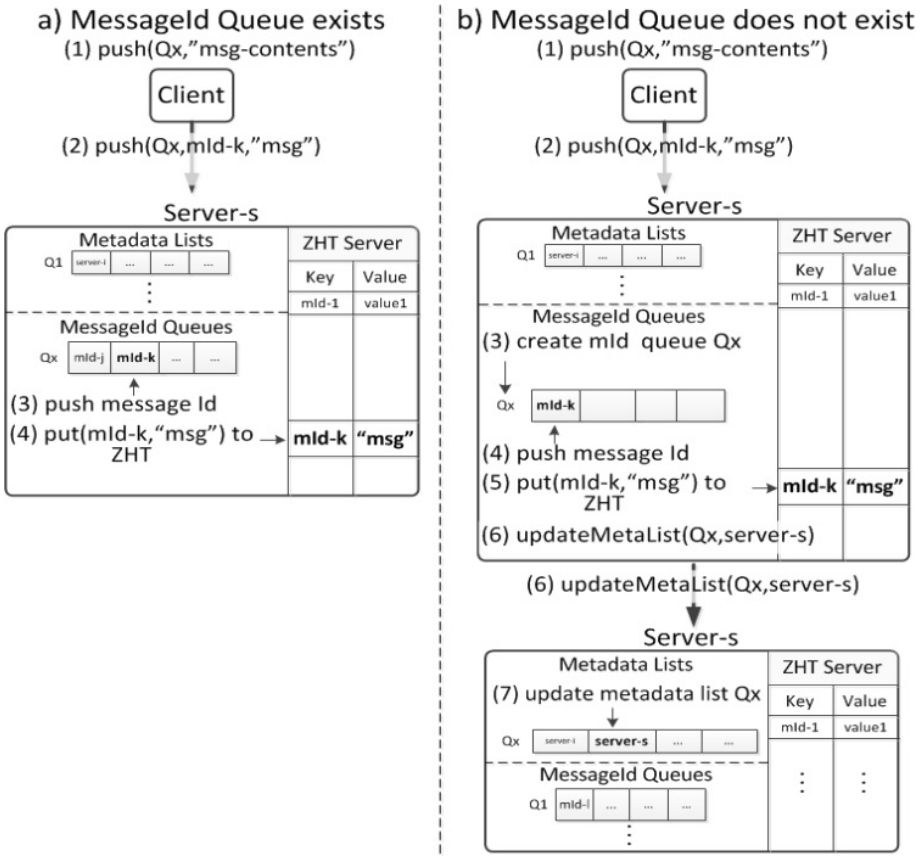

Figure 38. Push operation.

a) If the MessageId queue exists in this server, it will add the new MessageId to the queue and then it will make a put request to the underlying ZHT server. Since the hashing function used to hash the message Id on the client side is the same as the ZHT server's hashing function, the hash value will again determine the local server itself as the destination. Thus the ZHT server will add the message to its local NoVoHT server and there will be no additional network communications involved.

b) If the destination server does not have a MessageId queue with the name of this user queue, the server first creates a new MessageId queue for the user queue on this server, and then it will push the message to the MessageId queue and the local NoVoHT.

Meanwhile, the Metadata List of this user queue has to be updated with the information of the new server that keeps its messages. The server makes a request to the server that keeps Metadata List of the user queue and adds its own address to that list. The address of the destination server that keeps the Metadata list will be retrieved by hashing the name of the user queue.

3) pop

The pop method requests a message from a user queue on a local or remote Fabriq server. We want to make sure to retrieve a message from a Fabriq server with the lowest latency and the minimum network communication overhead.

A message of a certain queue may reside in any of the servers. The client can always refer to the Metadata list of a certain queue to get the address of a server that keeps messages of that queue. However, referring to the owner of the Metadata list in order to find a destination server adds network communication overhead and degrades the performance. Moreover, on larger scales, accessing the single metadata list owner could become a bottleneck for the whole system. In order to avoid the communication overhead, the client first tries the following ways before directly going to the metadata List owner:

(1) When a client starts to run, it first checks if there is a local Fabriq server running on the current node. The pop method first gets all of the messages on the local Fabriq server. The method sends the pop request to the local server and keeps getting messages until the mId queue is empty. After that the server returns a null value meaning that there is nothing left for this user queue on this server.

(2) After getting the null value, the client uses the second approach. It generates a random string and makes a pop request to a random server based on its hash value. Please note that the random string is not used as the message Id to be retrieved and it is only used to choose a remote server. If the destination server has messages, the client saves the random string as the last known server for the later accesses of this user queue. The client keeps popping messages from the last known server until it runs out of the messages for this user queue and returns null value.

(3) Finally, after client finds out that the last know server has returned null, using the hash value of the user queue name, it sends a request to the metadata list and gets the address of a server that has messages for this queue. Once a server returns null, the client again goes back to the metadata list owner and asks for a new server address.

Figure 39 shows a remote pop operation that only takes 1 hop. On the server side, the pop method looks for the MessageId queue of the requested user queue: **a)** If the mId queue does not exist in this server or if it is empty, the pop method returns a null value to the client. **b)** If the mId queue exists and has at least one message Id, it will retrieve a mId from the queue and makes a ZHT get request. Since the message Ids on the local queue have the same hash value as the local server's Id, the get request which is supposed to hash the message Id to find the server's address will get the value from the local ZHT server. Then the pop method will return that message to the client. If the retrieved mId was last one on the mId queue, the server calls a thread to asynchronously update the Metadata List of this user queue and remove the server Id from it.

4.2.3 Features

In this section, we discuss about some of the important features of Fabriq that makes it superior to other state-of-the-art message queues.

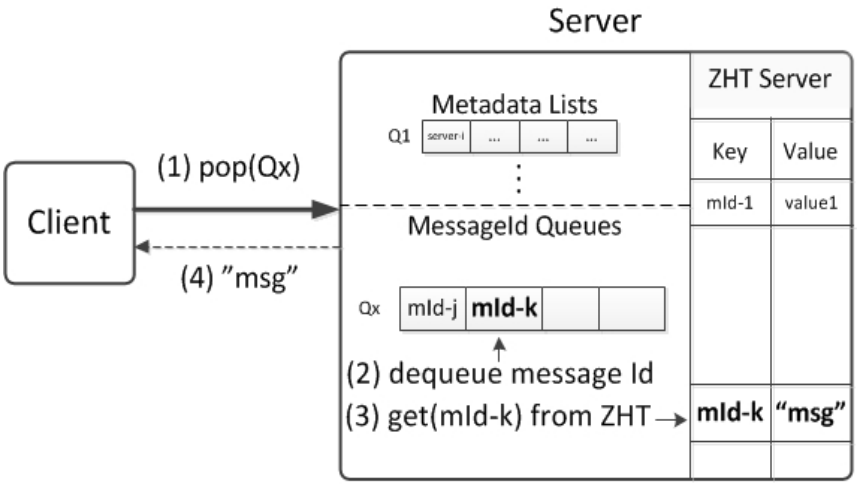

Figure 39. A remote pop operation with a single hop cost.

a) Load balancing

One of the design goals of Fabriq is to achieve a near perfect load balance. We want to make sure that the load from multiple queues gets distributed on all of the servers.

The load balancing of a system can highly depend on its message routing strategy. The systems with deterministic message routing usually have a static load distribution. That means the messages of multiple queues are statically split among all of the servers. This design is more convenient for the centralized and hierarchical architectures. However, there are many limitations with such design. In these architectures, the load balance on the system can fluctuate depending on the submission rate on different queues. On the other hand, the systems with non-deterministic routing have a more dynamic load on the servers. In order to have a dynamic load distribution, Fabriq Client

generates a randomly generated key for each message. Based on the hash value of the key, the message will be sent to a Fabriq server. Fabriq uses a uniformly distributed hash function.

b) Order of messages

Like many other distributed message queues, Fabriq cannot guarantee to keep the order of messages in the whole queue [46][23]. However, it can guarantee the order of the messages in a single server. The messages of a user queue are written in a local message queue on each server and the order of the messages is kept in that queue. Therefore the order of messages delivery in the server will be kept.

The message delivery order can be important for some workflows in scientific applications or HPC, we have provided a specific mode to define queues in a way that it keeps the message order. In this mode the messages of the user queue are only submitted to a single server. Since the order is kept in the single server, the order of the delivery will be kept as submitted.

c) Message delivery guarantee

When it comes to large scale distributed systems, the delivery of the content becomes a real challenge. Distributed systems cannot easily deal with this problem. On most of the loosely coupled systems where each node controls its own state, the delivery is not guaranteed. In distributed Message Queues, the delivery of the messages is an inevitable requirement. Therefore, most of the state of the art systems guarantee of the delivery. It is hard for the independent servers to synchronize with each other at large scales. Therefore

they guarantee the delivery at the cost of producing duplicate messages. Thus, they guarantee at least once delivery. Similarly in Fabriq, we guarantee at least once delivery.

Fabriq benefits from using a persistent DHT as its backbone. The push method in Fabriq makes a put request on the ZHT. The data in each server is persistent. ZHT also provides replication. Replication can prevent the loss of data in case of losing the hard disk on a node. The push and pop functions are both blocking functions. The client only removes the message from the memory after the server returns a success notification. There are two possible scenarios in case of the network or the server failure. If the message somehow does not get delivered, the server will not send a success notification. Therefore the push function times out and the message will be sent again. However, there is a possibility that the message gets delivered and the notification signal gets lost. In such scenario, the client will again send the message. This behavior could lead to duplicate messages on the system. However, the message destination is determined by the hash value of its message Id (mId). That means a message with a certain mId will always deliver to the same destination server. In case of the duplicate delivery, the ZHT destination server will notice a rewrite on the same Id and throws an exception. Therefore the messages are pushed with no duplicate messages.

Likewise, on the pop operation, the server only removes the message from ZHT when the client returns a success notification. The pop method first performs a ZHT get on the server side. It only performs a ZHT remove after it gets a notification from the client. If the client fails to receive the message or if it fails to notify the server, the message will remain at the server. Obviously, if the delivery happens with an unsuccessful server

acknowledgement, the server will keep the message. The same message which is now a duplicate message will be delivered on a later pop. Therefore Fabriq can only guarantee at least once delivery. But the difference of Fabriq with Kafka and SQS is the fact that it may only generate duplicate messages at the message pick up. The delivery of the message to the servers will not cause generating any duplicates.

d) Persistence

Fabriq extends the ZHT's persistence strategy to provide persistence. In ZHT, a background thread periodically writes the hash table data into the disk. Using ZHT, we can make sure the messages are safe in the hash table. But Fabriq still needs to keep its own data structure persistent in the disk. Otherwise, in case of system shut down of memory fail, Fabriq will not be able to retrieve messages from the hash table. In order to save the MessageId Queues and the Metadata List on each server, we have defined a few key-value pairs in the local NoVoHT table of each server. We save the list of the Metadata Lists and the MessageId Queues in two key-value pairs. We also save the contents of each single queue or list on an object and save those separately in the hash table. The background thread periodically updates the values of the data structures on the hash table. In case of failure, the data structures could be rebuilt using the key for the list of queues and lists in the hash table.

e) consistency and fault tolerance

Fabriq extends the ZHT strategies for its fault tolerance and consistency. It supports a strong consistency model on the first two replicas. The consistency is weak after the second replica. Fabriq also implements the lazy tagging of failed servers. In case of

failure the secondary replica will take over the delivery. The metadata lists and the MessageId queues of each Fabriq server are locally saved on its ZHT. Therefore they are automatically replicated on different servers. In case of the failure of a server, they can be easily regenerated from the replica server.

Another strategy which helps Fabriq provide better fault tolerance is spreading each user queue over all of the servers. In case of the failure of a server, without any need to link the client, the client will randomly choose any other server and continue pushing/retrieving messages from the system. Meanwhile, the secondary replica takes over and fills the gap.

f) Multithreading

Fabriq supports multithreading on the client side. The client can do push or pop using multiple threads. On the server side, Fabriq can handle simultaneous requests. But it does not use multithreading. The Fabriq server uses an event-driven model based on epoll which is able to outperform the multithreaded model by 3x. The event-driven model also achieves a much better scalability compared to the multithreading approach [23].

4.3 Network Communication Cost

In order to achieve low latency and high efficiency, it is important to keep the number of network communications low. In Fabriq, we design our push and pop operation with the minimum possible number of network communications. In this work, we consider each network communication as one hop.

Push cost: As shown in Figure 38, the push operation takes only one hop to complete. Since the update of the metadata list is executed by a separate thread in a non-blacking

fashion, we don't count it as an extra hop. Moreover, it only happens in the first push of each server. Therefore it does not count as an extra hop for the push operation.

Pop cost: A pop operation communication cost varies depending on the situation of both the client and the server. In order to be able to model the cost, we make a few assumptions and simplify our model. We assume that the uniform hash function works perfectly, and evenly distributes the messages among all of the servers. We analyze this assumption in practice in a later section.

We model the total cost of the pop operation in a system with a single consumer and multiple servers. *s* shows the number of servers and *m* shows the total number of messages that was produced by the clients. We model the cost in two situations: (a) when the total number of messages is more than the number of servers (*m*>*s*); and (b) when the number of messages is less than the number of servers (*m*<*s*). The total cost when *m*>*s* is shown below:

$$Total\ Cost_{(m>s)} = \frac{m}{s}\times 0 + \left(\frac{m}{s} - 1\right)(s-1)\times 1 + \sum_{i=1}^{s-1}\frac{i\times 3 + (s-i)\times 1}{s}$$

Based on the assumption of having perfect uniform distribution, we can assume that each server has *m/s* messages at the beginning of the consumption. Since the consumer first consumes all of the messages on its local server, the cost of the first *m/s* messages is going to be zero hop. After that, the consumer randomly chooses a server among the *s-1* that are left. The cost of finding a server with messages can be either 1 or 3. The client saves the id of the last known server and only makes a random call when the last known server has no messages left. After finding a new server, the client fetches all of the

messages on the last known server until the server is empty. The cost of all of these messages ((*m/s)-1*) is 1 hop. This process continues until all of the messages of each server are consumed. We can conclude that on each of the *s-1* remote servers there will be a single message that is going to be retrieved with the cost of 1 or 3 hops and (*m/s)-1* messages that are retrieved with the cost of exactly 1 hop. Having the total cost of the retrieval, we can calculate the average cost of each pop operation by dividing the total cost by the number of total messages:

$$Average\ Cost_{(m>s)} = 1 - \frac{1}{s} + \frac{s-1}{2m}$$

We can induce the range of the cost from the average cost formula. The average cost ranges from <1 to <1.5 hops. In the second scenario where the total number of messages is less than the number of servers, the total cost is:

$$Total\ Cost_{(m<s)} = \sum_{i=1}^{m} \frac{(s+i-(m+1))\times 3 + ((m+1)-i)\times 1}{s}$$

In this case, since each server gets one message at most, the cost of retrieving each message can be either 1 or 3. The average cost analysis is provided below:

$$Average\ Cost_{(m<s)} = 3 - \frac{m+1}{s}$$

Again, we can induce that the average cost of pop in this case ranges from 2 to 3 hops.

In order to confirm our analysis, we ran an experiment with a single consumer and counted the average number of hops on each pop operation. 0 shows the hop count in an experiment with 1 client and 64 servers. The total number of messages in this run was

64,000 messages. The results show that there were 1,079 messages on the local queue with the cost of 0 hops. Based on the cost model the average cost of hops in this experiment is 0.984 and the actual average cost is 1.053 hops, which means the model is fairly accurate.

The maximum communication cost in a system with multiple clients could be more than 3 hops. Since multiple clients can request a queue metadata owner for a message server at the same time, there is a chance that they both receive the same message server address from the metadata owner. Assuming the message server has only 1 message for this queue, the first client can get that last message, and the second client gets a null return value. In that case the client has to request the owner server again for another message server. This process can be repeated for *s* times until the client gets a message. However the chances of this occasion are very low. In fact, we have ran experiments in up to 128 instances scale and have not experienced a pop operation with more than 5 hops.

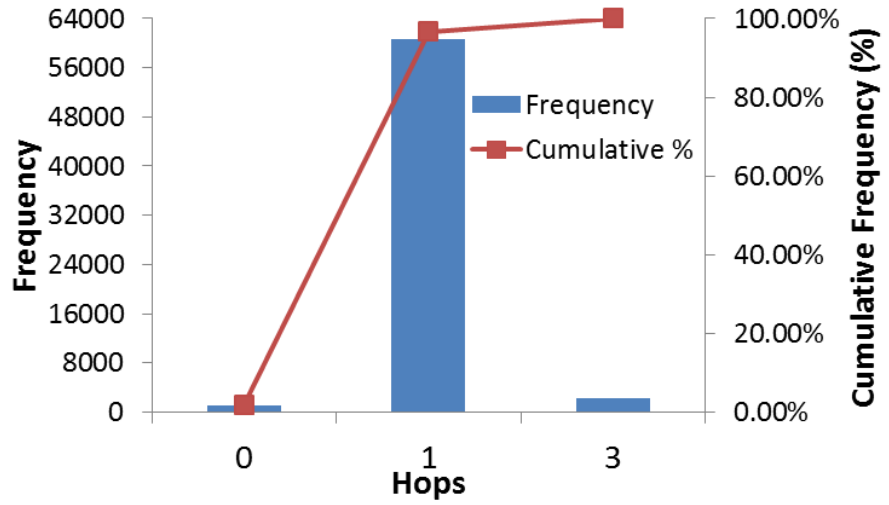

Figure 40. Cumulative Distribution of 1 client and 64 servers.

4.4 Performance Evaluation

This section analyzes the performance of Fabriq in different scenarios, compared with the other state of the two art Message Queue systems. But first, we summarize different features of Fabriq, compared with Kafka and SQS. We compare the performance of the three systems in terms of throughput and latency. We also compare the load balancing of the Kafka and Fabriq.

4.4.1 Fabriq, Kafka, and SQS

All three of the compared systems are fully distributed and are able to scale very well. However, they use different techniques in their architecture. Fabriq uses a DHT as its building block, while Kafka uses ZooKeeper [86] to handle the metadata management. SQS is closed source and there is minimal information available about its architecture.

One of the important features of a distributed queue is its message retrieval policy. All of the Fabriq servers act as a shared pool of messages together. That means all of the clients have equal chance of accessing a message at the same time. This feature enables the system to provide better load balancing. Moreover, having this feature, the producer can make sure that its messages are not going only to a specific consumer, but all of the consumers. SQS provides this feature as well. In Kafka, messages that reside in a broker (server) are only consumed by a single consumer at a time. The messages of that broker will only be available when the consumer gets the number of messages it requires. This can cause load imbalance when there is not enough messages in all of the brokers and degrade the system performance. This design goal in Kafka was a tradeoff to provide the rewind feature. Unlike other conventional queue systems including Fabriq and SQS,

Fabriq provides message rewind feature that lets consumers to re-consume a message that was already consumed. However, s mentioned before, having this feature means only one consumer can access a broker at a time.

Table 4. summarizes the features of the three queuing services. Unlike the other two systems, Fabriq does not support message batching yet. However this feature is currently supported in the latest version of ZHT and can be easily integrated with Fabriq. We expect that batching is going to improve the throughput significantly.

In Kafka brokers, messages are written as a continuous record and are only separated by the offset number. This feature helps Kafka provides better throughput for continues log writing and reading from producers and consumers. However, as mentioned before, this makes it impossible for multiple consumers to access the same broker at the same time. SQS and Fabriq save messages as separate blocks of data that enables those to provide simultaneous access on a single broker. All three of the systems provide the queue abstraction for multiple clients. In Fabriq and SQS, the client can achieve this by creating new queues. In Kafka, the client achieves this by defining new topics. Another important feature of Fabriq is the fact that it is able to run on different types of supercomputers including Blue Gene series that don't support Java. Kafka is written in Java, and SQS is closed source. Scientists are unable to use those two systems for HPC applications that run on such supercomputers.

Table 4. Comparison of Fabriq, SQS and Kafka

Feature	Fabriq	Kafka	SQS
Persistence	Yes	Yes	Yes
Delivery Guarantee	At least Once	At least Once	At least Once
Message Order	Inside Node	Inside Node	-
Replication	Customizable	Mirroring	3x
Shared Pool	Yes	No	Yes
Batching	No (Future work)	Yes	Yes

4.4.2 Testbed and configuration

Since SQS runs on AWS, in order to keep our comparisons fair, we chose Amazon EC2 as our testbed. The experiments scale from 1 to 128 instances. We chose m3.medium instances. Each instance has a single CPU core, a 1 Gigabit network card, and 16 GB of SSD storage.

4.4.3 Load balance

As discussed before, we believe that Fabriq provides a very good load balance. In this section we compare the load balancing of Fabriq with Kafka by checking the message distribution on the server of both systems. Since we don't have access to the servers on SQS, we cannot include this system on this experiment.

Figure 41 shows the number of messages received on each server of the two systems. In this experiment, each producer has sent 1000 messages. The total number of messages is 64000. The results show a very good load balance on Fabriq. The number of messages

range from 940 to 1088 messages on 64 servers. We ran the experiment 5 times and found out that the error rate is less than 5% for at least 89% of the servers, and is less than 9.5% in worst case. In Kafka, we observe a major load imbalance. The number of messages per server ranged from 0 to 6352. More than half of the servers got less than 350 messages.

Considering the fact that each server can only be accessed by one consumer at a time, we can notice that there will be a major load imbalance in the system. In a system with a 1 to 1 mapping between the servers and the consumers, more than half of the consumers go idle after finishing the messages of the underutilized servers and will wait for the rest of consumers to finish consuming their messages. Only after that, they can consume the rest of the messages and finish the workload.

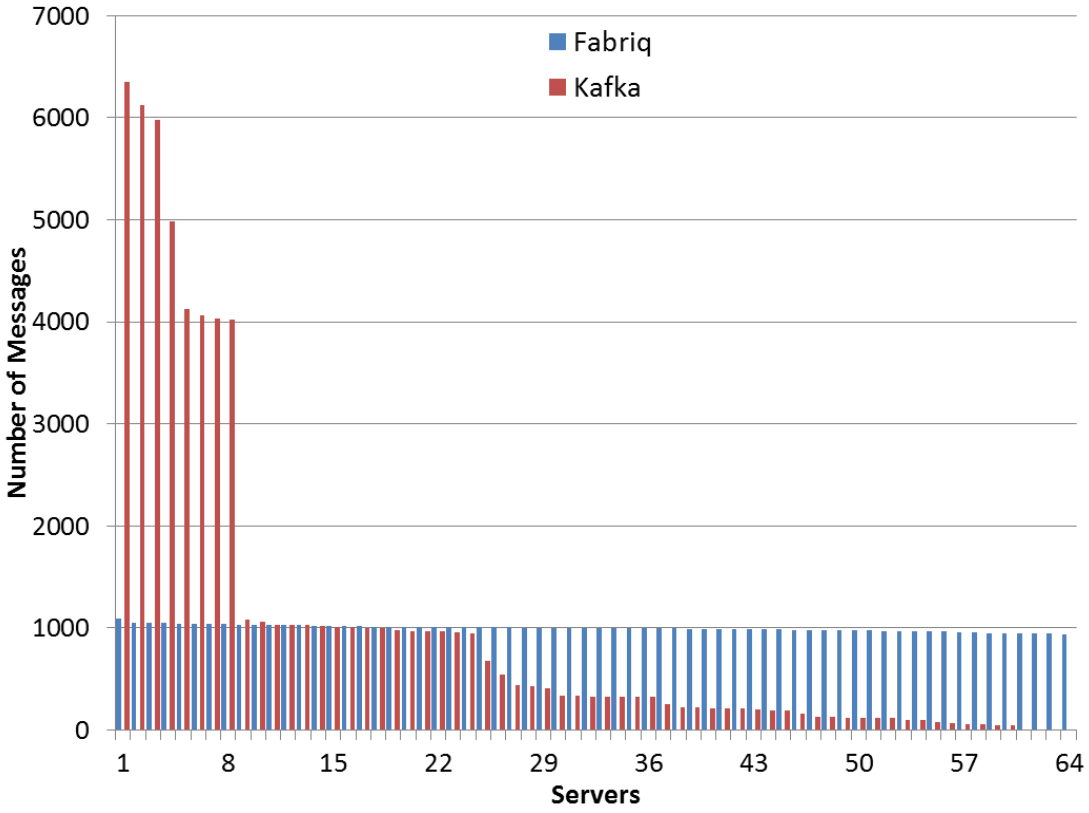

Figure 41. Load Balance of Fabriq vs. Kafka on 64 instances.

4.4.4 Latency

The latency of the message delivery is a very important metric for a distributed message queue. It is important for a DMQ to provide low latency on larger scales in order to be able to achieve high efficiency. Nowadays, many of the modern scientific and data analytics applications run tasks with the granularity of sub-seconds [14]. Therefore, such systems will not be able to exploit a message queue service that delivers messages in the order of seconds.

We measured latency by sending and receiving 1000, 50 bytes messages. Each instance ran 1 client and 1 server. Figure 42 shows the average latency of the three systems in push and pop operations.

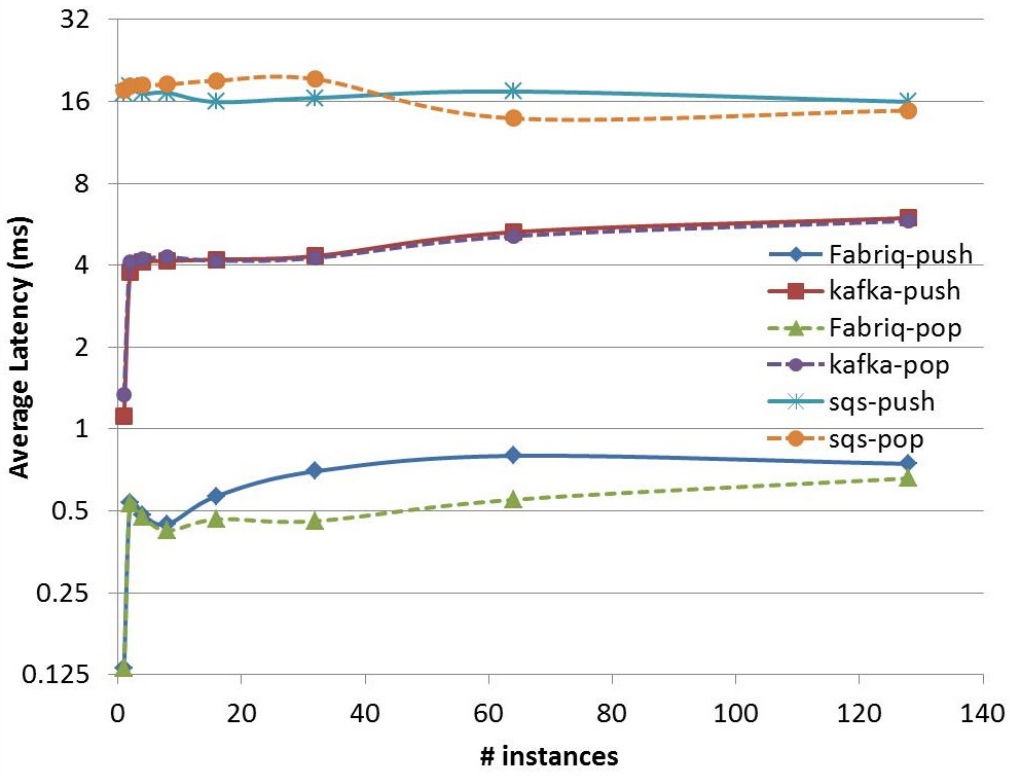

Figure 42. Average latency of push and pop operations

All the three systems show stable latency in larger scale. Fabriq provides the best latency among the three systems. Since the communications are local at the scale of 1 for Kafka and Fabriq, they both show significantly lower latency than the other scales. We can notice that there is almost an order of magnitude difference between the average latency of Fabriq and the other two systems. In order to find out the reason behind this difference, we have generated the cumulative distribution on both push and pop operations for the scales of 64 and 128 instances. According to Figure 43, at the 50

percentile, the push latency of Fabriq, Kafka, and SQS are respectively 0.42ms, 1.03ms, and 11ms. However, the problem with the Kafka is having a long tail on latency. At the 90 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 0.89ms, 10.4ms, and 10.8ms. We can notice that the range of latency on Fabriq significantly shorter than the Kafka. At the 99.9 percentile, the push latency of Fabriq, Kafka, and SQS are respectively 11.98ms, 543ms, and 202ms.

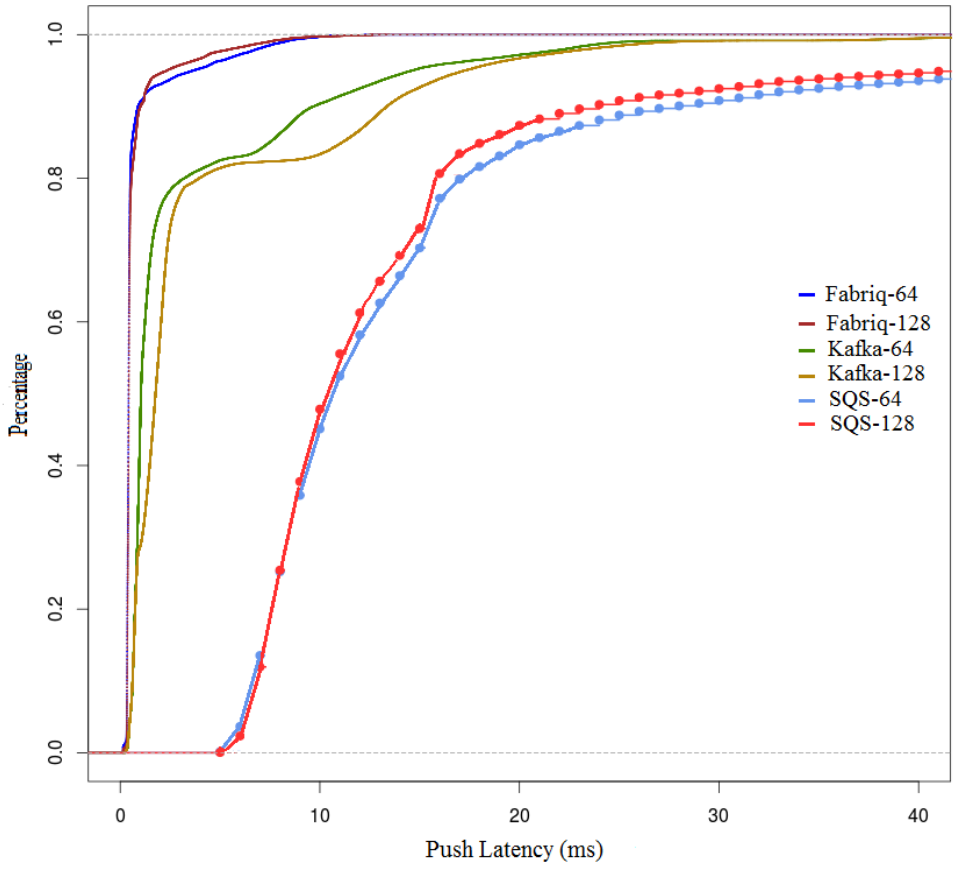

Figure 43. Cumulative distribution of the push latency.

Similarly, Figure 44 shows a long range on the pop operations for Kafka and SQS. The maximum pop operation time on the on Fabriq, Kafka, and SQS were respectively 25.5ms, 3221ms, and 512ms.

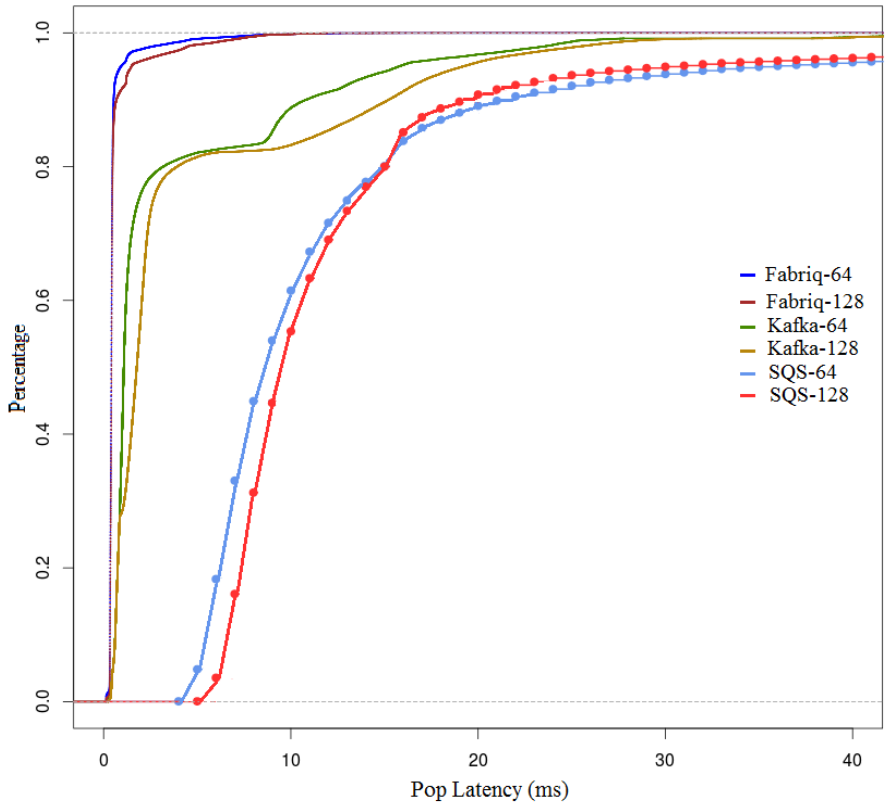

Figure 44. Cumulative distribution of the pop latency.

As we observed on from the plots, Fabriq provides a more stable latency with a shorter range than the other two systems. Among the three systems, Kafka has the longest range of latency. There could be many reasons for the poor performance of Kafka. Before starting to produce or consume, each node needs to get the broker information from a centralized ZooKeeper. In larger scales, this could cause a long wait for some of the nodes. Another reason for the long range of message delivery is the load imbalance. We have already discussed about it on the previous sections.

4.4.5 Throughput

It is substantial for a DMQ to provide high throughput in different scales. In this section, we compare the throughput of the three systems. We have chosen three different message sizes to cover small, medium and large messages. All of the experiments were run on 1 to 128 instances with a 1 to 1 mapping between the clients and servers in Fabriq and Kafka. In SQS, since the server is handled by AWS, we only run the client that includes producer and consumer on each instance.

Figure 45 shows the throughput of both push and pop operations for the short messages. Each client sends and receives 1000 messages that are each 50 bytes long. Among the three systems, Fabriq provides the best throughput on both push and pop operations. As mentioned before, due to problems such as bad load distribution, and the problem of single access to the broker by the consumers, the throughput of Kafka is almost an order of magnitude lower than the Fabriq.

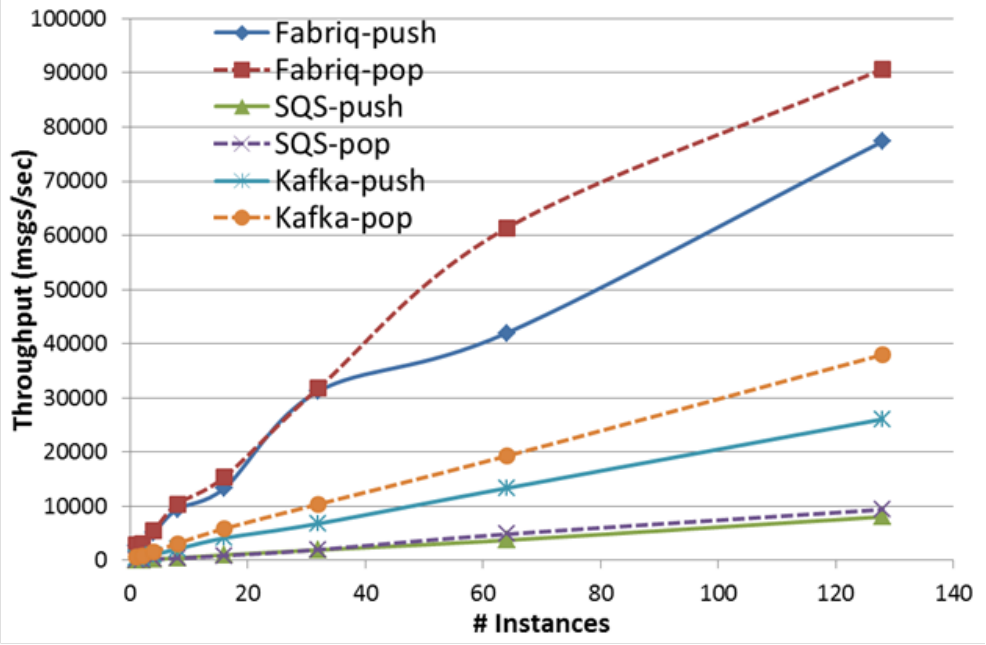

Figure 45. Throughput for short (50 bytes) messages (msgs/sec).

All of the three systems are scaling almost nearly up to the scale of 128 instances. We can also notice that the throughput of pop operation is higher than the push operation. The reason for that in Fabriq is that the consumers first try to fetch the local messages. Also, in general we know that in a local system, the read operation is usually faster than the write operation. Figure 46 compares the throughput of push and pop operations for medium (256KB) and large (1MB) messages. At the largest scale, Fabriq could achieve 1091 MB/sec on push operation and 1793 MB/sec on pop operation. We notice that the throughput of the Kafka for push and pop operations is respectively 759 MB/sec and 1433 MB/sec which is relatively close to what Fabriq can achieve. The reason for that is the continuous writing and reading on the same block of file instead of having separate files for different messages. This way, Kafka is able to deliver large messages with the minimum overhead. Therefore it performs well while delivering larger messages.

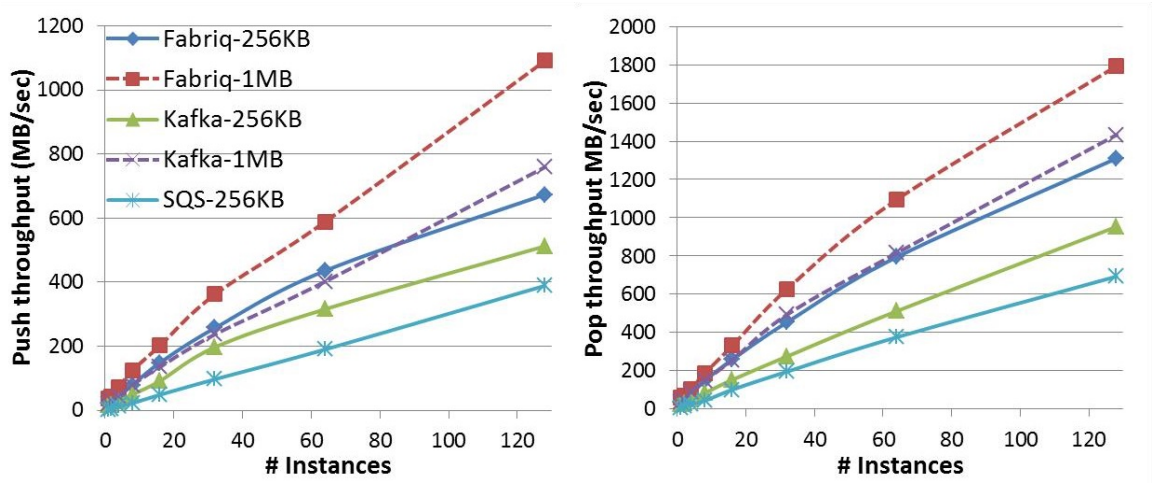

Figure 46. Push and pop throughput for large messages (MB/sec).

4.5 Summary

A Distributed Message Queue can be an essential building block for distributed systems. A DMQ can be used as a middleware in a large scale distributed system that decouples different components from each other. It is essential for a DMQ to reduce the complexity of the workflow and to provide low overhead message delivery. We proposed Fabriq, a distributed message queue that runs on top a Distributed Hash Table. Fabriq was designed with the goal of achieving low latency and high throughput while maintaining the perfect load balance among its nodes. Servers in Fabriq are fully independent. The load of each queue is shared among all of the nodes of the Fabriq. This makes Fabriq achieve good load balance and high availability. The network communication protocol in Fabriq is tuned to provide low latency. A push operation could take 0 to 1 roundtrip communication between the servers. A pop operation takes 0, 1 or 3 operations for more than 99% of the operations.

The results show that Fabriq achieve higher efficiency and lower overhead than Kafka and SQS. The message delivery latency on SQS and Kafka is orders of magnitude larger than Fabriq. Moreover, they have a long range of push and pop latency which makes them unsuitable for applications that are sensitive to operations with long tails. Fabriq provides a very stable latency throughout the delivery. Results show that more than 90% of the operations take less than 0.9ms and more than 99% percent of the operations take less than 8.3ms in Fabriq. Fabriq also achieves high throughput is large scales for both small and large messages. At the scale of 128, Fabriq was able to achieve more than 90000 msgs/sec for small messages. At the same scale, Fabriq was able to deliver large messages at the speed of 1.8 GB/sec.

There are many directions for the future work of Fabriq. One of the directions is to provide message batching support in Fabriq. The latest version of ZHT which is under development supports message batching. We are going to integrate Fabriq with the latest version of ZHT and enable the batching support. Another future direction of this work is to enable our network protocol to support two modes for different workflow scenarios. In this feature, the user will be able to choose between the two modes of heavy workflows with lots of messages, and a moderate workflow with less number of messages. We are going to optimize Fabriq for task scheduling purposes and leverage it in CloudKon [46] and MATRIX [92] which are both task scheduling and execution systems optimized for different environments and workflows. Finally, inspired by the work stealing technique used in MATRIX [92], we are planning to implement message-stealing on the servers in order to support pro-active dynamic load balancing of messages. Pro-active load

balancing of the messages helps balancing the server loads when the message consumption is uneven.

CHAPTER 5

ALBATROSS: AN EFFICIENT CLOUD-ENABLED TASK SCHEDULING AND EXECUTION FRAMEWORK USING DISTRIBUTED MESSAGE QUEUES

Data Analytics has become very popular on large datasets in different organizations. It is inevitable to use distributed resources such as Clouds for Data Analytics and other types of data processing at larger scales. To effectively utilize all system resources, an efficient scheduler is needed, but the traditional resource managers and job schedulers are centralized and designed for larger batch jobs which are fewer in number. Frameworks such as Hadoop and Spark, which are mainly designed for Big Data analytics, have been able to allow for more diversity in job types to some extent. However, even these systems have centralized architectures and will not be able to perform well on large scales and under heavy task loads. Modern applications generate tasks at very high rates that can cause significant slowdowns on these frameworks. Additionally, over-decomposition has shown to be very useful in increasing the system utilization. In order to achieve high efficiency, scalability, and better system utilization, it is critical for a modern scheduler to be able to handle over-decomposition and run highly granular tasks. Further, to achieve high performance, Albatross is written in C/C++, which imposes a minimal overhead to the workload process as compared to languages like Java or Python.

As previously mentioned, the goal of this work is to provide a framework that could efficiently handle resources at large scales and is a good fit for cloud environment. After analyzing the unique features and requirements of the cloud environment, we designed and implemented the building blocks for a framework that is able to schedule tasks and

job at large scale in cloud. In this chapter, we propose Albatross, a task level scheduling and execution framework that uses a Distributed Message Queue (DMQ) for task distribution among its workers. Unlike most scheduling systems, Albatross uses a pulling approach as opposed to the common push approach. The former would let Albatross achieve a near perfect load balancing and scalability. Furthermore, the framework has built in support for task execution dependency on workflows. Therefore, Albatross is able to run various types of workloads, including Data Analytics and HPC applications. Finally, Albatross provides data locality support. This allows the framework to achieve higher performance through minimizing the amount of unnecessary data movement on the network.

5.1 Background and Motivation

The massive growth in both scale and diversity of Big Data has brought new challenges as industry expectations of data processing loads continue to grow. For example, more than 2.5 exabytes of data is generated everyday [1]. At various organizations, it has become a necessity to process data at massive scales. However, processing data at such scales is not feasible on a single node, therefore it is vital to utilize distributed resources such as Clouds, Supercomputers, or large clusters. Traditionally, these clusters are managed by batch schedulers, such as Slurm [4], Condor [6], PBS [7], and SGE [8]. However, such systems are not capable of handling data processing at much larger scales. Another pitfall these systems must face is processing finer granular tasks (far more in number, much shorter run times) at larger

scales. This could only be handled with a sophisticated scheduler capable of handling many more tasks per second.

The above mentioned systems were designed for clusters with far fewer amounts of batch jobs that usually run for a longer time. Those batch jobs usually have a different nature than the data analytics jobs. Moreover, they all have centralized designs that make them incapable of scaling up to the needs of today's data analysis.

Data analytics frameworks such as Hadoop [91] and Spark [100] were proposed to particularly solve the problem of data processing at larger scales. These frameworks distribute the data on multiple nodes and process it with different types of tasks. However even these frameworks have not been able to completely solve the problem. Both of the above mentioned systems have centralized bottlenecks that make them unable to handle the higher rate of task and data volume. Therefore, these frameworks are not suitable for workloads that generate more tasks in shorter periods of times. To give an example, simple applications, such as matrix multiplication, may be embarrassingly parallelizable, while generating many tasks where each task occurs in a very small amount of time [53]. Our evaluations show that Spark and Hadoop are not able to schedule and execute more than 2000 tasks per second which could add significant overhead to those applications.

Other frameworks like Sparrow [14] have tried to bypass the issue of the centralized architecture on Spark and Hadoop. Their solution is to primarily dedicate each job that consists of multiple tasks to a separate scheduler. This solution raises a few issues. First, the utilization of the whole cluster will be lower than distributed task level scheduling solutions. Since different jobs have different sizes, they will cause load imbalance for the

system, and since a scheduler can only handle its own job, an idle or lightly loaded scheduler will not be able to help any overloaded schedulers. Moreover, this solution may not work well for the jobs that have significantly higher number of heterogeneous tasks. Such a job could easily saturate a single centralized task scheduler and cause significant overheads to the system.

Nowadays, most of the data analytics of big data run on the Cloud. Unlike HPC Clusters and Supercomputers that have homogeneous nodes, Clouds have heterogeneous nodes with variable node performance. Usually the underlying physical hardware is being shared across multiple Virtual Machines (VMs) and subsequently, these nodes may have variable performance [67]. Our evaluations have shown that in some cases, identical instances on AWS [15] within a given region and availability zone can exhibit variable performance results. This means some tasks can take much longer than others. It is important for a scheduler to take this into the consideration. A simple and effective solution would be breaking tasks into smaller tasks and make them more granular. This technique is called over-decomposition. If the tasks are more granular, the workload can be better spread over the nodes and more capable nodes (faster, less-utilized) will be able to run more tasks. This would allow system utilization to significantly increase [119]. However, this poses significant challenges to the scheduling system, forcing it to make faster scheduling decisions. To allow over-decomposition and handle finer granular task scheduling, it is essential for modern schedulers to provide distributed scheduling and execution at the task level rather than the job level.

It is also critical for a scheduler to impose minimal overhead to the workload execution process starting from a single node. Tasks in utilizing a fine granular workflow could take a few milliseconds of execution time. It is not practical to run such workloads on a scheduler that takes seconds to schedule and execute a single task. Some programming languages (e.g. Java and Python) that operate at a more abstract level could add more overhead to the scheduling process. Therefore it is necessary to implement the scheduler in lower level languages such as C or C++ to achieve the best performance on a single node level.

There is an emergent need for a fully distributed scheduler that handles the scheduling at the task level and is able to provide efficient scheduling for high granular tasks. In order to achieve scalability, it is important to avoid a centralized component as it could become a bottleneck. In this paper, we propose Albatross: A fully distributed cloud-enabled task scheduling and execution system that utilizes a distributed Message Queue as its building block.

The main idea of scheduling in Albatross is to use Fabriq, which is a distributed message queue for delivering tasks to the workers in a parallel and scalable fashion. Most of the commonly used schedulers have a central scheduler or a controller that distributes the tasks by pushing them to the worker nodes. However, unlike the traditional schedulers, Albatross uses a pulling approach as opposed to pushing tasks to the servers. The benefit of this approach is to avoid the bottleneck of having a regional or a central component for task distribution. Albatross also uses a Distributed Hash Table (DHT) for the metadata management of the workloads. There is no difference between any of the

nodes in Albatross. Each node is a worker, a server, and possibly a client. The DMQ and the DHT are dispersed among all of the nodes in the system. The task submission, scheduling, and execution all happen through the collaboration of all of the system nodes. This feature enables Albatross to achieve a high scalability. The communication and the routing of the tasks all happen through hashing functions that have an O(1) routing complexity. That makes the communications between the servers optimal.

Albatross is able to run workflows with task execution dependency through a built in support in the DMQ. That gives Albatross flexibility to run HPC, and data analytics jobs. An HPC job is usually defined as a Bag-of-Tasks [121] with dependencies between those tasks. The built-in task dependency support will enable the application to submit jobs to Albatross without having to provide an application level support for task dependencies. DAG support also enables Albatross to run various types of data analytics workloads. The focus of this paper is mainly Map-reduce workloads.

Another important feature that is required for data analytics frameworks is data locality support. Data locality suggests that since the movement of the data on the network between the nodes is an expensive process, the frameworks have to prioritize moving tasks to the data location and minimize the data movement on the system. In Albatross this feature is supported through load balancers of the Fabriq.

Our evaluations show that Albatross outperforms Spark and Hadoop that are currently state-of-the-art scheduling and execution frameworks for data analytics in many scenarios. It particularly outperforms the other two when the task granularity increases. Albatross is able to schedule tasks at 10K tasks per second rate, outperforming Spark by

10x. The latency of Albatross is almost an order of magnitude lower than Spark. Albatross's throughput on real applications has been faster than the two other systems by 2.1x and 12.2x. Finally, it outperforms Spark and Hadoop respectively by 46x, and 600x in processing high granularity workloads on grep application.

In summary, the main contributions of Albatross are:

- The framework provides a comprehensive workload management including: data placement and distribution, task scheduling, and task execution.
- It has a fully distributed architecture, utilizing a DMQ for task distribution and a DHT for workload metadata management.
- It provides distributed scheduling at the task level, as opposed to job level distributed scheduling.
- The framework provides an efficient Task execution dependency support. It enables Albatross to run a wide range of workloads including HPC and Data Analytics.
- It provides data locality optimization.
- It offers an optimized implementation for High Performance Applications, using C/C++ programming language.

Before discussing Albatross, we are going to provide information about the two building blocks of the framework. Fabriq and ZHT are the main components of the Albatross that make the task distribution, communication and the metadata management possible in this framework.

5.1.1 ZHT overview

For metadata management, Albatross uses ZHT which is a low overhead and low latency Distributed Hash Table, and has a constant routing time. It also supports persistence. ZHT has a simple API with 4 major methods: insert, lookup, remove, and append. A key look up in ZHT can take from 0 (if the key exists in the local server) to 2 network communications. This helps provide the fastest possible look up in a scalable DHT. The following sections discuss main features of ZHT. ZHT operations are highly optimized and efficient. *Insert* and *lookup* operations take less than a millisecond on an average instance on AWS.

1) Network Communication

ZHT supports both TCP and UDP protocols. In order to optimize the communication speed, the TCP connections are cached by a LRU cache. That will make TCP connections almost as fast as UDP.

2) Consistency

ZHT supports consistency via replication. In order to achieve high throughput ZHT follows a weak consistency model after the first two replicas that are strongly consistent.

3) Fault Tolerance

ZHT supports fault tolerance by lazily tagging the servers that are not being responsive. In case of failure, the secondary replica will take the place of the primary replica. Since each ZHT server operates independently from the other servers, the failure of a single server does not affect the system performance.

4) Persistence

ZHT is an in-memory data-structure. In order to provide persistence, ZHT uses its own Non-Volatile Hash Table (NoVoHT). NoVoHT uses a log based persistence mechanism with periodic check-pointing.

5.1.2 Fabriq overview

Fabriq is a Distributed Message Queue that runs on top of ZHT. It was originally designed for handling the delivery of high message volumes on Cloud environment. Adding an abstract layer over ZHT, Fabriq is able to provide all of the benefits of it including persistence, consistency, and reliability. Running on top of a DHT, Fabriq is able to scale more than 8k-nodes.

Messages in Fabriq get distributed over all of its server nodes. Thus, a queue could coexist on multiple servers. That means clients can have parallel access to a queue on Fabriq, making Fabriq a perfect fit for our framework. Albatross uses a DMQ as a big shared pool of tasks that could provide simultaneous access from a large number of its workers.

Fabriq guarantees exactly-once delivery of the messages. That is an important requirement for Albatross. Since CloudKon was using SQS, and SQS could generate duplicate messages, we had to add an extra component to filter the duplicate messages. That adds more overhead to the system. Using Fabriq, we will not have that problem since we can make sure that it only delivers a message once. Fabriq is very efficient and scalable. The latency of pop and push operations on Fabriq over an Ethernet network on AWS are less than a millisecond on average. That is almost an order of magnitude better than other state-of-the-art message queues.

5.2 System Overview

Albatross is a distributed task scheduling and execution framework. It is a multipurpose framework, suitable for various types of workloads and compatible with different types of environments, especially the Cloud environment. The key insight behind the Albatross is that unlike other conventional schedulers, in Albatross, a worker is the active controller and the decision making component of the framework. In most of the schedulers, there is a central or regional component that is responsible for pushing tasks to the workers and keeping them busy. That could bring a lot of challenges in many cases. In the case of running workloads with high task submission rates, or workloads with heterogeneous tasks, or in the case of larger scale systems, or heterogeneous environments like Clouds, these schedulers will show significant slowdowns. In such schedulers, the scheduler component needs to have live information about the workers that are being fed. That could be a big bottleneck and a source of long delays as the component can get saturated after a certain scale or under a certain load. Albatross gets rid of central or regional scheduler components by moving the responsibility from the scheduler to the workers. In Albatross, the workers pull tasks from a shared pool of tasks whenever they need to run a new task. That could significantly improve the utilization and could improve the load balancing of the system. In order to achieve that, Albatross uses Fabriq as a big pool of tasks. Fabriq is scalable and provides parallel access by a large number of workers. That makes Fabriq a perfect fit for Albatross.

5.2.1 Architecture

In order to achieve scalability, it is inevitable to move the control from centralized or regional components to the workers in the framework. In such architecture, the scheduling, routing, and task execution take place by the collaboration of all of the nodes in the system. Each worker has the same part in the workload execution procedure and there is not a single node with an extra component or responsibility.

Figure 47 shows the components of Albatross. The system is comprised of two major components, along with the worker and client driver programs. Fabriq is responsible for the delivery of the tasks to the workers. ZHT is responsible for keeping the metadata information and updates about the workload. This information could include the data location, and the workload DAG. Depending on the setup configurations, an Albatross node could run a worker driver, a client driver, a ZHT server instance, a Fabriq server instance, or a combination of those components. Since ZHT and Fabriq are both fully distributed and do not have a centralized component, they can be distributed over all of the nodes of the system. Therefore, each worker driver program has local access to an instance of Fabriq server, and an instance of ZHT server.

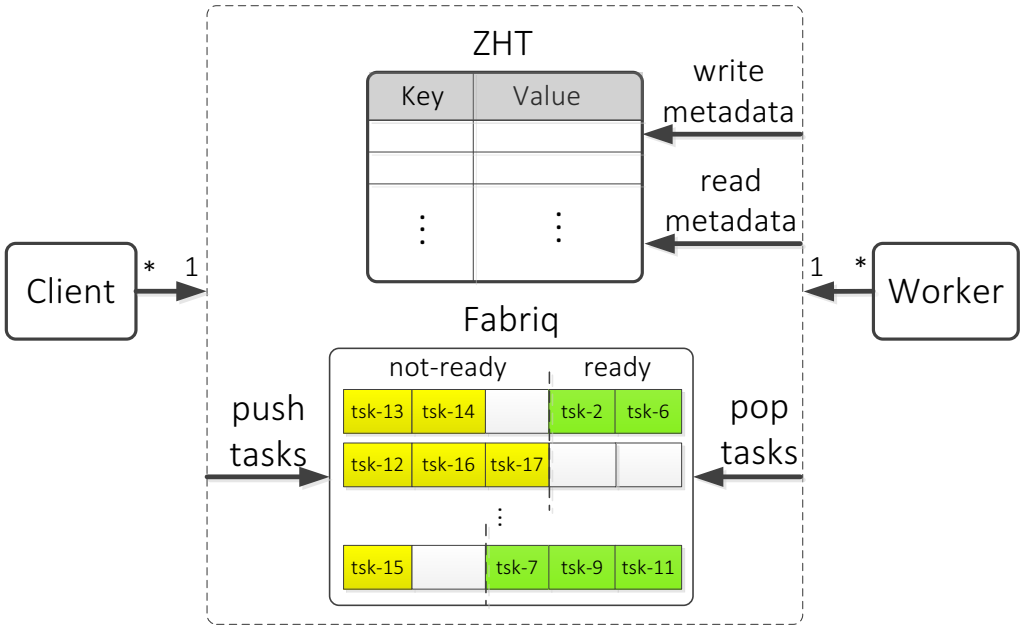

Figure 47. Albatross Components

The process starts from the Client driver. The user provides the job information, and the input dataset to the client program. Based on the provided job information, the Client program generates the tasks and the workload DAG. Then it distributes the dataset over all of the worker nodes in the system. Finally it submits the tasks to Fabriq. The task submission on Fabriq is performed via a uniform hashing function that distributes the tasks among all the servers, leading to a good system load balance. The data placement and the task submission could also be performed through multiple parallel clients.

The Worker driver starts with pulling tasks from Fabriq. Depending on the type of the task and the workload, a Worker might access, or write into ZHT to either get metadata information about the input data location, or dependencies, or to update the metadata. The worker also fetches the input data either locally or from a remote Worker. We discuss the

procedure of data locality support on Albatross in the following sections. The output of each task is written locally.

5.2.2 Task execution dependency

Both HPC and data analytics workloads enforce a certain execution order among their tasks. In order to be able to natively run those workloads, Albatross needs to provide support for workload DAGs. HPC tasks propose the concept the Bag-of-Tasks. Each HPC job could have some tasks that could be internally dependent on each other. Data analytics workloads often propose similar concepts. Programming models such as Dryad [116], and Map-reduce that are often used in data analytics have similar requirements. In Map-reduce, reduce tasks could only run after all of their corresponding map tasks have been executed.

The implementation of the task execution dependency support should not disrupt the distributed architecture of the system. Our goal was to keep the dependency support seamless in the design and avoid adding a central component for keeping the DAG information. Task execution dependency is supported through the implementation of priority queues in Fabriq. Each task in Albatross has two fields that hold information about its execution dependencies. ParentCount (pcount) field shows the number of unsatisfied dependencies for each task. In order to be executed, a task needs to have its ParentCount as 0. ChildrenList field keeps the list of the taskIDs of the current task's dependent tasks.

Figure 48 shows the process of running a sample DAG on Albatross. The priority inside Fabriq has two levels of priorities: 0 or more than 0. The priority queue inside each

Fabriq server holds onto the tasks with non-zero pcounts. A worker can only pop tasks with 0 pcounts. Unlike conventional DMQs, Fabriq provides ability to directly access a task via its taskID. That feature is used for Albatross task dependency support. Once a task is executed, the worker updates its ChildrenList tasks, decreasing their dependencies by 1. Inside the priority queue, once a task's pcount becomes 0, the queue automatically moves it to the available queue and the task could be popped by a worker.

5.2.3 Data locality

Data locality aims to minimize the distance between data location and respective task placements. Since moving the data is significantly more expensive than moving the process, it is more efficient to move the tasks to where the data is located. Data locality support is a requirement for data-intensive workloads. Albatross's goal is to minimize the data movement during the workload execution while maintaining high utilization.

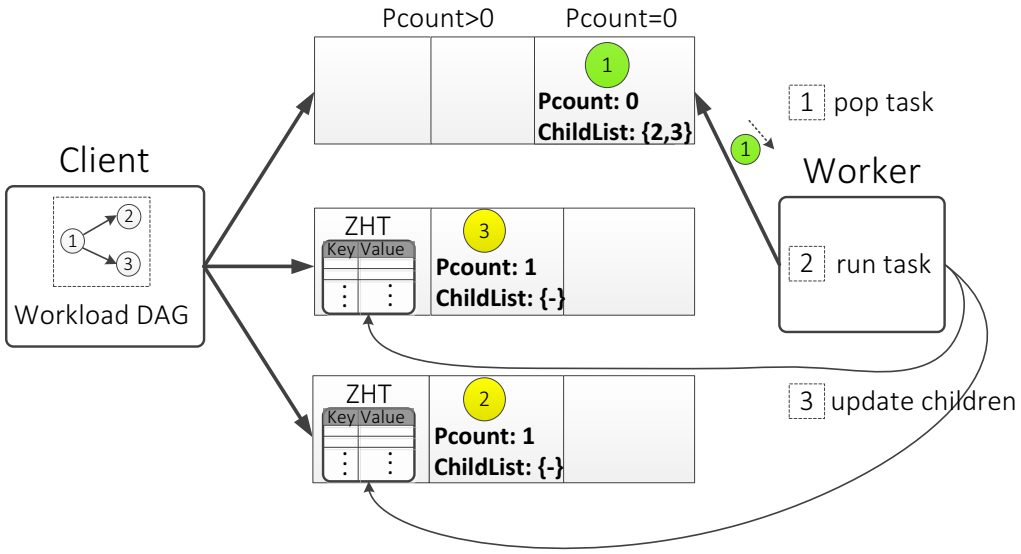

Figure 48. Task Execution Dependency Support

Data locality could be applied on different stages of the workload execution process. The common approach is to dictate the decisions on the scheduling and task placement stage. On this approach, the scheduler tries to send the tasks to their corresponding data location. This approach minimizes the number of task placements before a task gets executed. However, there are some issues with this approach that is going to hurt the overall performance of the system at larger scales. In order to send the tasks to the right location, the scheduler needs to have extensive information about the workers, and the data locations. That could slow down the process at larger scales. Moreover, this approach could lead into load imbalance and reduce the utilization as there could be some nodes with many tasks and some left idle because their corresponding tasks are dependent on the current running tasks on the workload. Also, this method is associated with the pushing approach which is not desirable at larger scales.

Albatross does not dictate any logics regarding data locality at task submission stage. It uses a uniform hashing function to distribute them evenly among servers. That means the task placement is going to be random. The data locality is achieved after the tasks are submitted to the Fabriq servers. Depending on the location of their corresponding data, some of the tasks might be moved again. Even though that adds extra task movement to the system process, it lets the workers handle the locality between themselves without going through a single centralized scheduler. Figure 49 shows the data locality process and its corresponding components. There are two types of queues and a locality engine on each Albatross server. The main queue belongs to the Fabriq. It is where the tasks first land once they are submitted by the client. The locality engine is an independent thread

that goes through the tasks on main queue and moves the local tasks to the local queue. If a task is remote (i.e. the corresponding data is located on another server), the engine sends the task to the local queue of its corresponding server via that server's locality engine.

Strict dictation of data locality could not always be beneficial to the system. Workloads usually have different task distribution on their input data. A system with strict data locality always runs the tasks on their data local nodes. In many cases where the underlying infrastructure is heterogeneous, or when the workload has many processing stages, a system with strict locality rules could have a poorly balanced system where some of the servers are overloaded with tasks and the rest are idle with no tasks to run. In order to avoid that, we incorporate a simple logic in the locality engine.

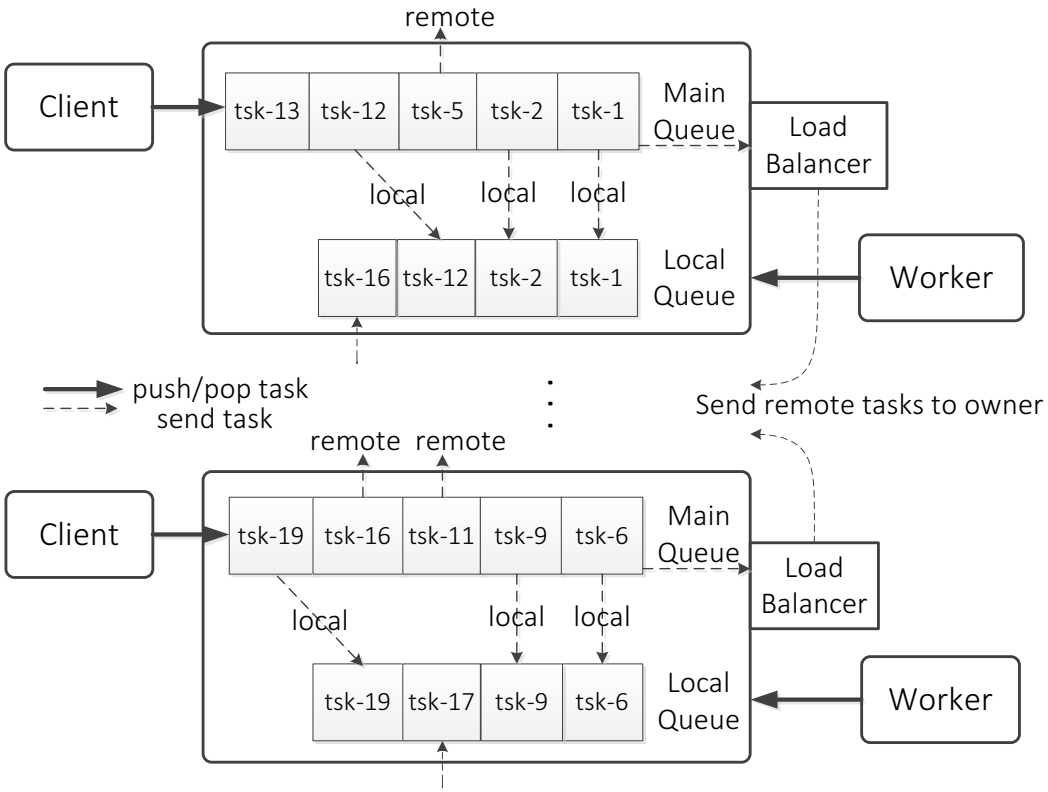

Figure 49. Data Locality Support Components

Figure 50 portrays the locality engine's decision making process. When the engine finds a remote task, it tries to send it to the remote server. If the remote server is overloaded, it rejects the task. In that case, the engine saves the task to the end of local queue regardless of being remote. The worker is going to pop tasks from the local queue one by one. Once it wants to pop the remote task, the locality engine tries to send the task to its own server one more time. If the remote server is still overloaded, the locality engine transfers the corresponding data from the remote server. This technique is similar to the late-binding technique that is used in other scheduling frameworks [14].

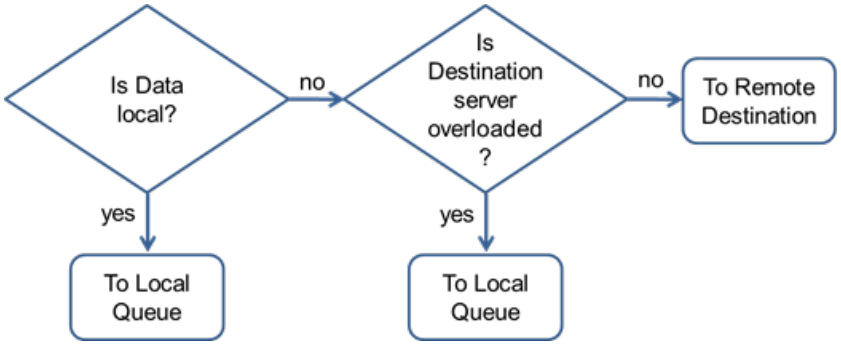

Figure 50. Locality Engine's Decision Making

5.3 Map-Reduce on Albatross

This section discusses the implementation of the Map-reduce programming model in Albatross. Figure 51 shows the Map-Reduce execution process. Like any other Map-reduce framework, the process starts with the task submission. A task could be either map or reduce. Map tasks have their pcount as 0. They usually have a reduce task in ChildrenList. Reduce tasks have their pcount as more than 0 and will be locked in the queues until their parent map tasks are executed.

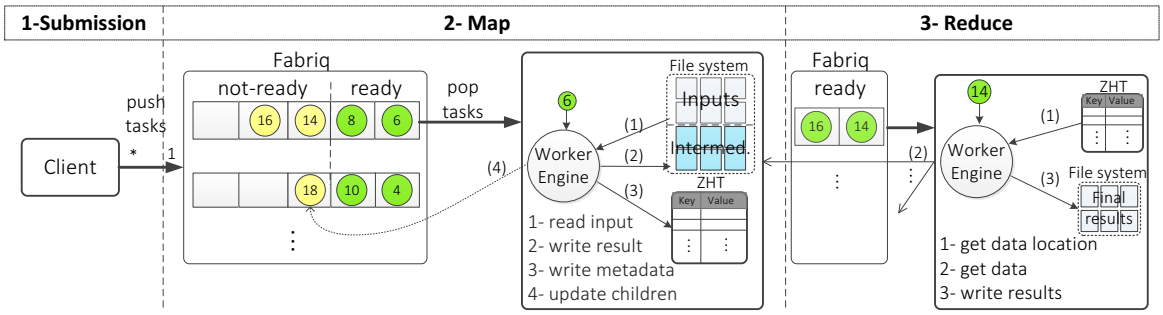

Figure 51. Map-reduce process in Albatross

Once a map task is popped by a worker, it loads its input data from the local (or remote) file system. The map function is included inside the map task. Mapper loads the function and runs it. Unless the size of the output exceeds the available memory limit, the

worker writes the intermediate results to the memory. For applications like sort that have larger intermediate data, the worker always writes the output on disk. Once the task is executed, the worker adds the location of the intermediate data for this map task to the ZHT. Finally, the pcount for its dependent reduce task will be reduced by 1.

Once all of the parent map tasks of a reduce task are executed, the reduce task becomes available and gets popped by a worker. The worker gets the location of all of the intermediate data required by this task. Then the worker gets the intermediate data from those locations. Then it loads the reduce function from the task and writes the final results to its local disk. It also adds its final results location to the ZHT.

Many Map-reduce frameworks including Hadoop and Spark have many centralized points in their process. The mappers and the reducers have to go through a single component to get or update information such as the intermediate data location, the logging information and the final output location. Albatross has no centralized point of process in its Map-reduce model. The tasks are delivered through Fabriq and the other information is propagated through ZHT. Since ZHT resides on all of the nodes, mappers or reducers could all access ZHT at the same time without causing a system slow down.

5.4 Performance Evaluation

This section analyzes the performance of Albatross. We compare the performance of Albatross using different metrics with Spark and Hadoop which are both commonly used for data analytics. First, we briefly compare the major differences of the three systems in design and architecture. Then, we compare the performance of those frameworks while running microbenchmarks as well as real applications. We measure the efficiency,

throughput, and latency of the three systems while varying the granularity of the workloads.

5.4.1 Hadoop, Spark and Flink

1) Hadoop

Hadoop is a data analytics framework that adopted its architecture from Google's Map-reduce implementation. It consists of two main components which are the distributed file system (HDFS) and the Map-reduce programming paradigm. The two main centralized components of Hadoop are the NameNode and the JobTracker. The JobTracker is in charge of tracking any disk reads/writes to HDFS. As the job tracker must be notified by the task trackers, similarly the namenode must be notified of block updates executed by the data nodes. In the newer version of Hadoop, instead of the job tracker as seen in Hadoop 1.x, there is a resource manager. The resource manager (similar to the job tracker) is in charge of allocating resources to a specific job [118]. Although the Yarn [117] version tries to provide higher availability, the centralized bottlenecks are still existent.

2) Spark

Spark, like many other state-of-the-art systems, utilizes a master-slave architecture. The flexible transformations enable Spark to manipulate and process a workload in a more efficient manner than Hadoop. One of the vital components of the Spark's cluster configuration is the cluster manager which consists of a centralized job-level scheduler for any jobs submitted to the cluster via the SparkContext. The cluster manager allocates a set of available executors for the current job and then the SparkContext is in charge of

scheduling the tasks on these allocated executors. Therefore, similar to Hadoop, centralized bottlenecks are still present even though the capability of iterative workloads is better handled in Spark than in Hadoop. The primary bottlenecks in Spark's cluster mode are the task level scheduler present in the SparkContext and the job-level scheduler present in the provided cluster manager. Other than these pitfalls, Spark provides a novel idea of resilient distributed datasets or RDDs which allows it to provide support for iterative workloads. RDDs are analogous to a plan or a series of transformations which need to be done on a set of data. Each RDD is a step and a list of these steps form a lineage.

3) Flink

Apache Flink is an in-memory distributed big data processing platform with goals similar to Hadoop and Spark . Flink was mainly designed to optimize the processing of distributed batch data and stream data [124]. The main component of Flink is the streaming dataflow engine which is responsible for data distribution, communication, and fault tolerance for distributed computations over data streams. Flink's execution engine treats batch data as a data stream with a finite dataset and time. The major novelty of Flink is using pipelines on its design that makes it a perfect fit for stream data [125]. Using pipelines and iteration operators enables Flink to provide native support for iterative applications [126]. Moreover, due to using pipelines, Flink is able to overlap between different stages.

Flink has been able to outperform frameworks like Spark and Hadoop due to its optimized architecture for streaming and batch data [127]. On iterative workloads, Spark

users need to generate an extra set of tasks for each iteration. However, in Flink there is no need for an explicit set of tasks for each iteration and the execution engine is able to natively run the multiple iterations until they meet a certain criteria defined by the user. Another important feature of Flink is its optimized memory management that makes it easier to work with compared to Spark. In case of working with large datasets that don't fit in memory, Flink and Albatross are able to spill the extra data from the memory to disk. However, Spark does not have that feature. That causes task out of memory errors which lead to task restarts. In Spark, user needs to explicitly set a data size threshold for the framework that specifies the maximum size of the data that could fit in memory. However, in Flink there is no need to explicitly set a threshold as the platform has ability to automatically spill that extra data to disk. During the shuffle stage, Flink is able to move the intermediate data directly from a nodes memory to another memory. However, in spark, in many cases the data transfer happens through disk which slows down the process.

Flink has many different modes for its job distribution. User can use Yarn, the Flink's default stand-alone scheduler, or Mesos for workload distribution. The main limitation of the Flink is the centralized architecture of its job distributor. All of the above mentioned options schedule and distribute jobs from centralized components. As discussed before on previous sections, the centralized architecture limits the scalability of the system. Moreover, due to using separate and independent pipelines Flink does not have a job level fault tolerance support. That means if a tasks fails, the whole job needs to be restarted.

5.4.2 Testbed and configurations

The experiments were done on m3.large instances which have 7.5 GB of memory, 32 GB local SSD storage, and Intel Xeon E5-2670 v2 (Ivy Bridge) Processor (2 vCores). Since the amount of vCores available were two, the number of reduce tasks for Hadoop was limited to only two concurrently running tasks.

For the overall execution time experiments (block size fixed at 128 MB), the workload was weakly scaled by 5 GB per added node. For the varied partition/block size experiments, the workload was weakly scaled by 0.5 GB per added node. There were two main reasons as to why the same 5GB per node workload was not used for the varied partition/block size experiments. Spark and Hadoop started seeing a very long execution time (~4 hours for a single node experiment) and Spark which uses the Akka messaging framework uses a framesize (pool) in which the completed tasks were being stored. As the HDFS block size decreased, the number of total tasks (total number of tasks = total workload / block size) increased to amounts which the default framesize configuration could not handle. Finally, regarding the microbenchmarks, instead of focusing on the size of the input data, we focused on the amount of tasks which should be run per node. As can be seen below, a total of 1000 tasks were run per node.

5.4.3 Microbenchmarks

This section compares the scheduling performance of the Albatross and Spark while running synthetic benchmarks. These benchmarks are able to reflect the performance of the systems without being affected by the workloads or applications. We measure latency and throughput while scheduling null tasks. We did not include Hadoop in this section, as

Hadoop's tasks are written to disk. That makes the scheduling significantly slower than the other two systems.

1) Latency

In order to assess the scheduling overhead of a framework, we need to measure overall latency of processing null tasks. In this experiment, we submit 10,000 null tasks per nodes and calculate the total time for each task. The total time could be defined as the time it takes to submit and execute a task, plus the time for saving the results on disk or memory. There is no disk access in this experiment.

Figure 52 shows the average latency of running empty tasks on the three frameworks, scaling from 1 to 64 nodes. Ideally, on a system that scales perfectly, the average latency should stay the same.

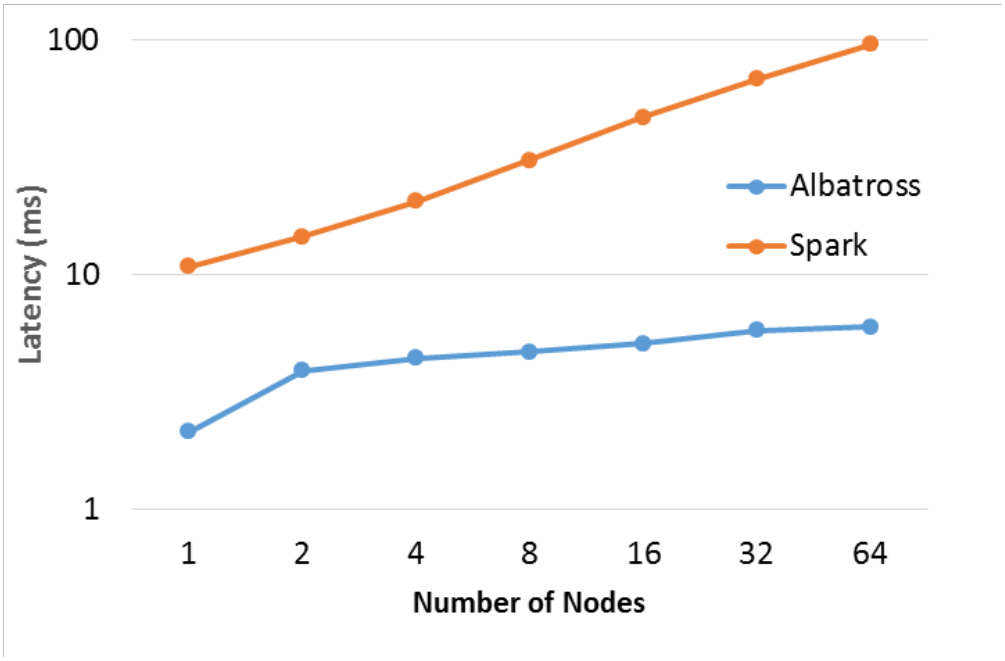

Figure 52. Average latency of in-memory null tasks

In order to fully reflect the scheduling overhead of three frameworks, we need to measure metrics like minimum, median, and maximum latency. Figure 53 shows the cumulate distribution function (cdf) of the three systems while running empty tasks. The cdf is able to show the possible long tail behavior of a system. Compared to Spark, Albatross has a much shorter range in scheduling tasks. The slowest task took 60 milliseconds to schedule. That is 33x faster than the slowest task in Spark which took more than 2 seconds. This long tail behavior could significantly slow down certain workloads. The median scheduling latency in Albatross is 5 ms as compared to 50 ms latency in Spark. More than 90% of the tasks in Albatross took less than 12 ms which is an order of magnitude faster than the Spark at 90 percentile.

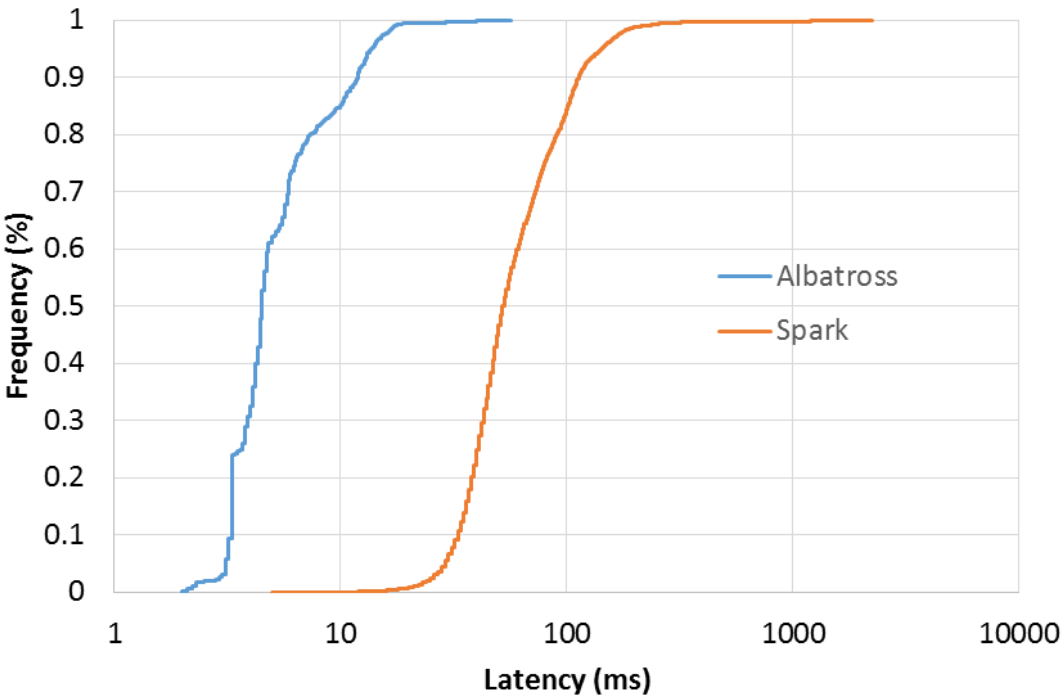

Figure 53. Cumulative distribution null tasks latency

2) Scheduling Throughput

In order to analyze the task submission and scheduling performance of the frameworks, we measured the total timespan for running 10,000 null tasks per node. The throughput is defined as the number of tasks processed per second (tasks per second).

Figure 54 shows the throughput of Albatross and Spark. Spark is almost an order of magnitude slower than Albatross, due to having a centralized scheduler, and being written in Java. Also, unlike Albatross the performance of Spark scheduling does not linearly increase with the scale. Spark's centralized scheduler gets almost saturated on 64 nodes with a throughput of 1235 tasks per second. We expect to see the Spark scheduler saturating at 2000 tasks per seconds. Albatross was able to linearly scale, reaching to 10666 tasks per second at 64 nodes scale.

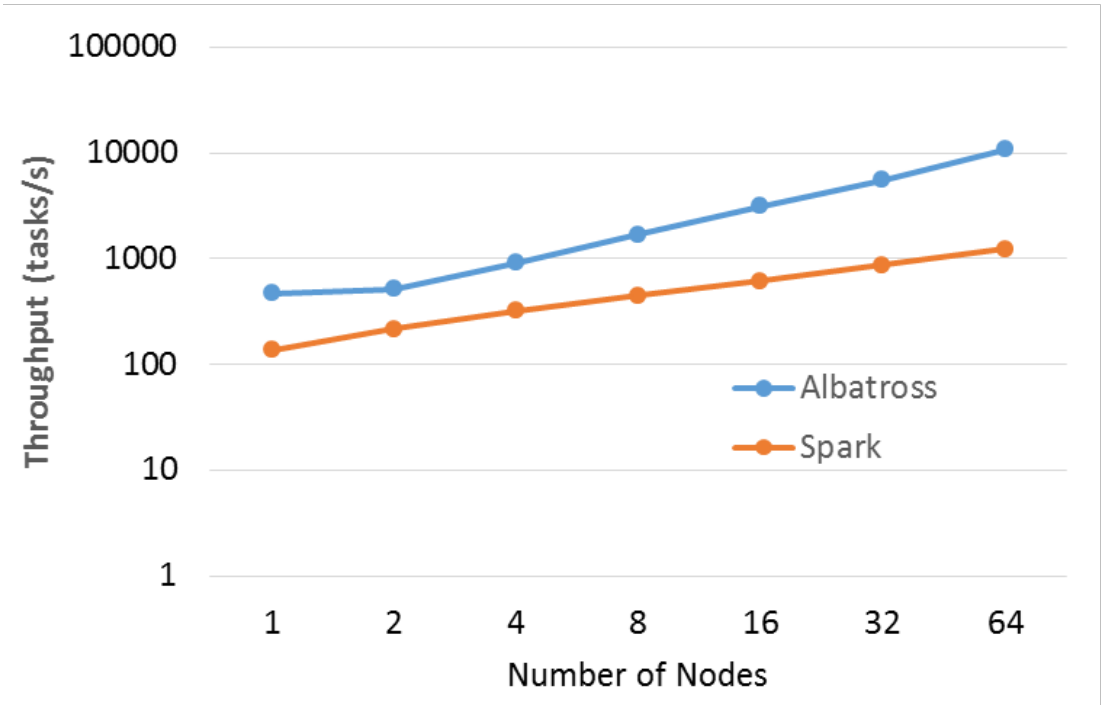

Figure 54. Throughput of null task scheduling

5.4.4 Application performance

In order to provide a comprehensive comparison, we compare the performance of the three frameworks while running different Map-reduce applications. Applications were chosen to reflect weaknesses and advantages of frameworks in different aspects. We have measured the performance for sort, -word-count, and grep applications.

1) Sort: The sort application sorts the input dataset lines according to the ascii representation of their keys. The algorithm and the logic of the sort application is based on the Terasort benchmark that was originally written for Hadoop [118]. Unlike the other two applications, intermediate data in sort is large and will not always fit in memory. The application performance reflects the file system performance and the efficiency of the memory management on each framework. Also, the network communication is significantly longer in this application. As portrayed in Figure 55, Spark utilizes a lazy evaluation for its lineage of transformations. Only when an action such as saveAsHadoopFile is received is when the entire lineage of transformations is executed and data is loaded into memory for processing. Therefore, sortByKey, which is the only transformation in this scenario has a relatively quick "execution time." On the other hand, the action phases require loading the data in memory, actually performing the sort, and writing back to disk.

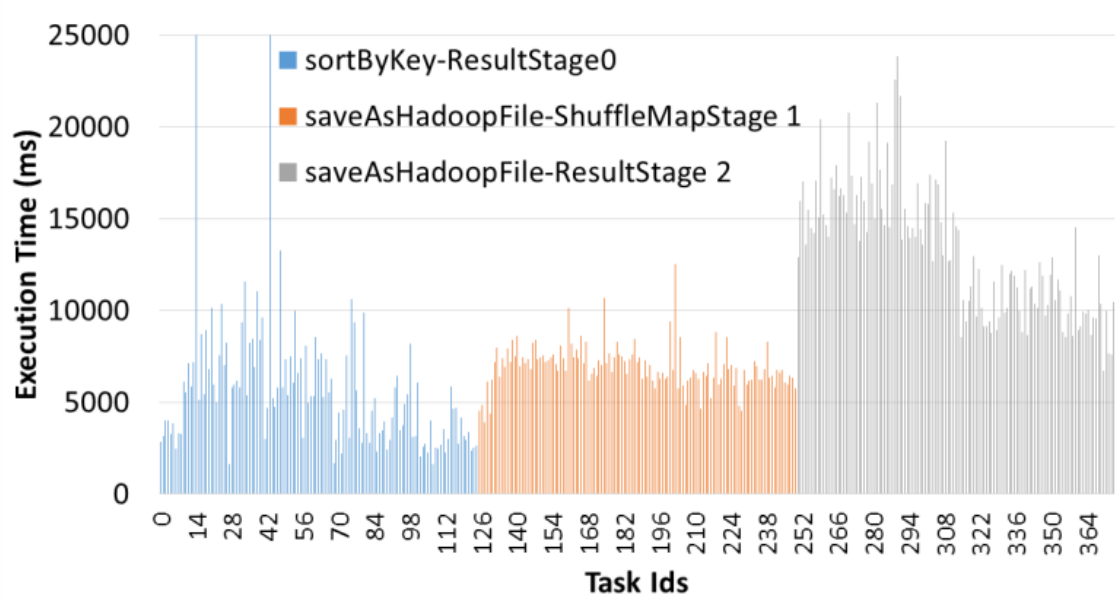

Figure 55. Spark's task breakdown (32 nodes)

As shown in Figure 56, Hadoop's sort has two phases for the Map-reduce model. The reduce phase is relatively longer since it includes transferring data and the sorting after receiving the data. To allow for Hadoop to utilize a similar "range partitioner" as Spark, we implemented Hadoop's sort using a TotalOrderPartitioner.

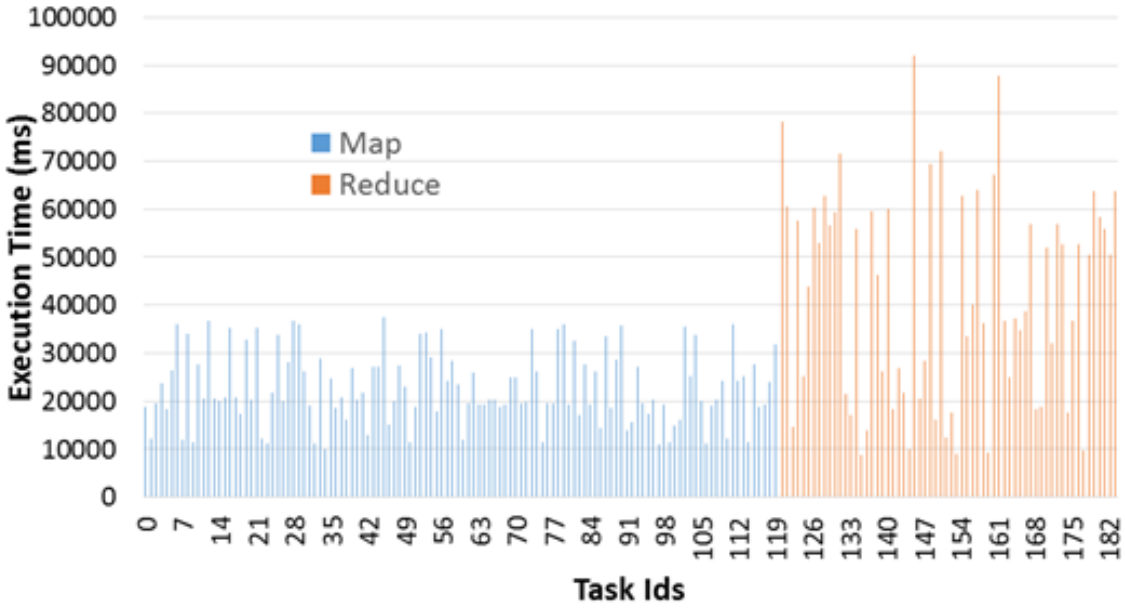

Figure 56. Hadoop's task breakdown (32 nodes)

Figure 57 portrays the map and reduce runtimes for sort application. Similar to Hadoop, Albatross has two phases for sort application. The runtime variation of tasks on Albatross is significantly lower than the Hadoop and Spark. This is a result of the pulling approach of Albatross that leads to a far better load balancing on the workers.

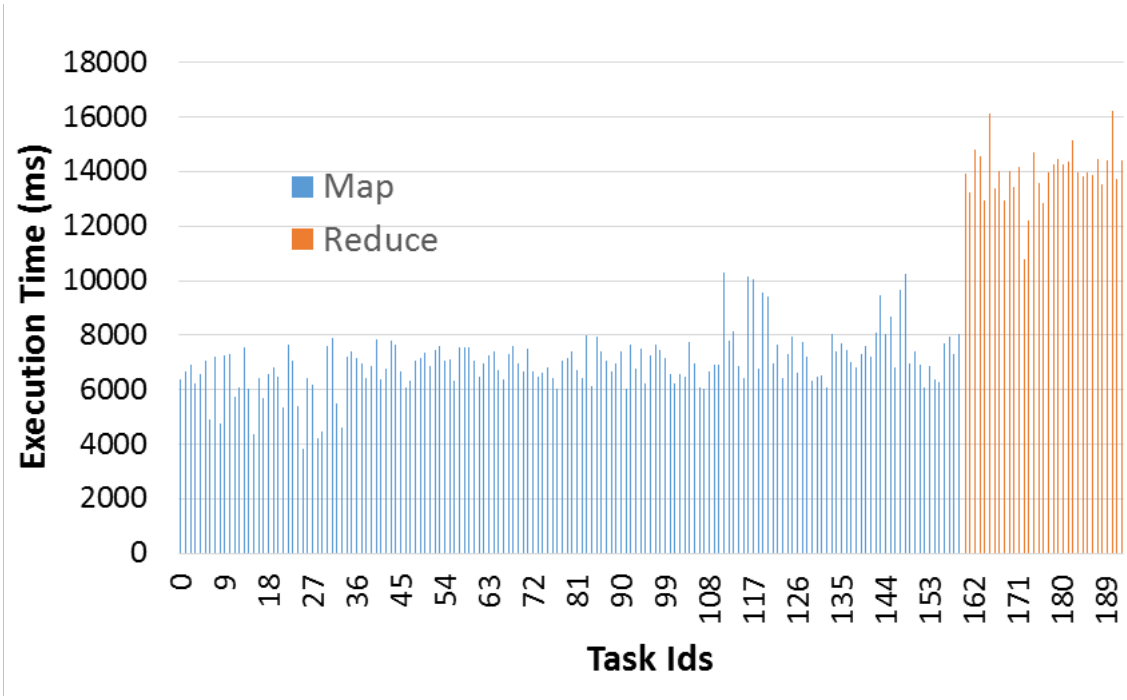

Figure 57. Task breakdown for Albatross (32 nodes)

2) Word-count: The word-count application calculates the count of each word in the dataset. Unlike sort, the proportion of intermediate data to input data is very low. Each map task generates a hash-map of intermediate counts that is not bigger than a few kilobytes. The intermediate results will get spread over all of reducers based on their key-ranges. Similar to Hadoop's word-count, Spark's word-count uses map tasks which output (word, 1) pairs and a reducer which aggregates all the (word, 1) pairs by key. The final action is a saveAsHadoopFile which saves the resulting pairs to a file on HDFS

3) Grep: The grep application searches for the occurrences of a certain phrase within the input dataset. The workflow process is similar to the behavior of Map-reduce. However, in Albatross, unlike Map-reduce, the intermediate result of each map task only moves to a certain reducer. That leads to far fewer data transfers over the network. In

order to send read-only data along with a task to the executors, Spark encapsulates the read-only data in a closure along with the task function. This is a simple, but very inefficient way to pass the data since all workers will have duplicate values of the data (even though the variable values are the same). Since the grep implementation needs each map task to have access to the search pattern, a broadcast variable which stored the four byte search pattern was used.

We compare the throughput of the frameworks while running the applications. We measure the total throughput based on the ideal block size for each Framework. We also analyze the performance of the frameworks while increasing granularity of the workloads. Figure 58 shows the performance of sort, scaling from 1 to 64 nodes. Albatross and Spark show similar performances up to 16 nodes. However, Spark was not able to complete the workload as there were too many processes getting killed, due to running out of memory. As we mentioned earlier, intermediate results in sort are as big as the input. Using Java, both Spark and Hadoop were not able to handle processing inputs as they were generating large intermediate results. There were too many task restarts on larger scales for Hadoop and Spark. The dotted lines are showing the predicted performance of the two systems if there were not running out of memory and processes were not getting killed by the Operating System. In order to avoid this problem, Albatross writes the intermediate results to disk when it gets larger than a certain threshold.

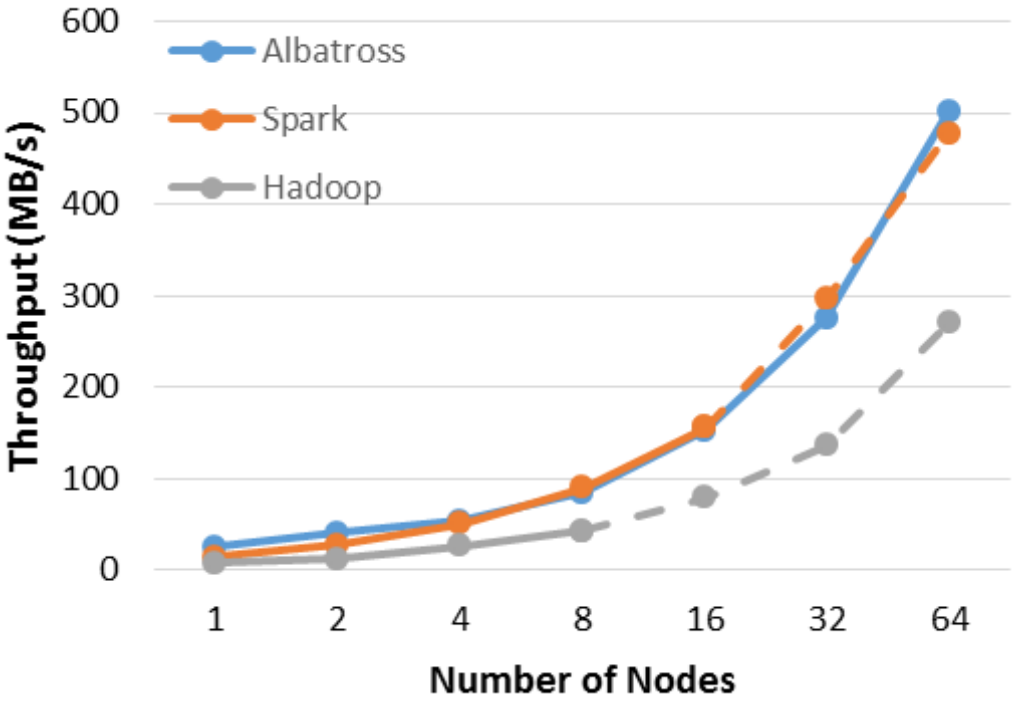

Figure 58. Throughput of sort application

Figure 59 shows the throughput of sort with different partition sizes on 64 nodes. The throughput of Spark is 57% of Albatross using 100MB partitions. However, this gap becomes more significant on smaller partitions. The throughput of Spark is less 20% of Albatross using 1MB partitions. This clearly shows the incapability of Spark on handling high granularity. On the other hand, Albatross proves to be able to handle over-decomposition of data very well. Albatross provided a relatively stable throughput over different partition sizes. The Albatross scheduling is very efficient and scalable and could handle higher task submission rate of the workload. The only exception was for the 100KB partitions. At 100KB, opening and reading files takes the majority of the time and

becomes the major bottleneck on the processing of each task. Hadoop and Spark cannot use partitions smaller than 1MB due to the limitation of HDFS.

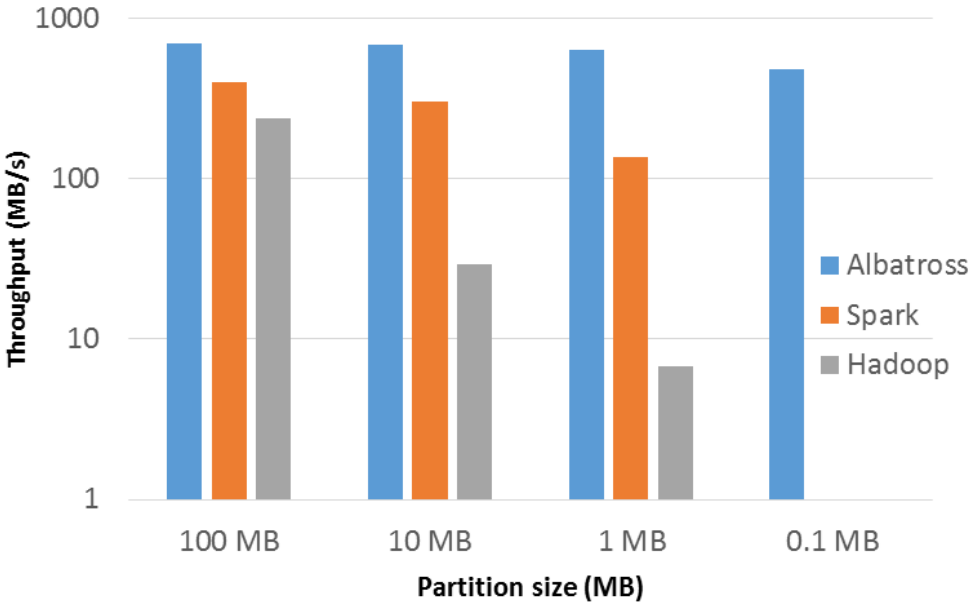

Figure 59. Sort application throughput (varied partition sizes)

Figure 60 shows the throughput for word-count using large data partitions. Spark was able to achieve a better throughput than the other two systems. Even though they have provided different throughputs, all the three systems linearly scaled up to the largest scale of the experiment.

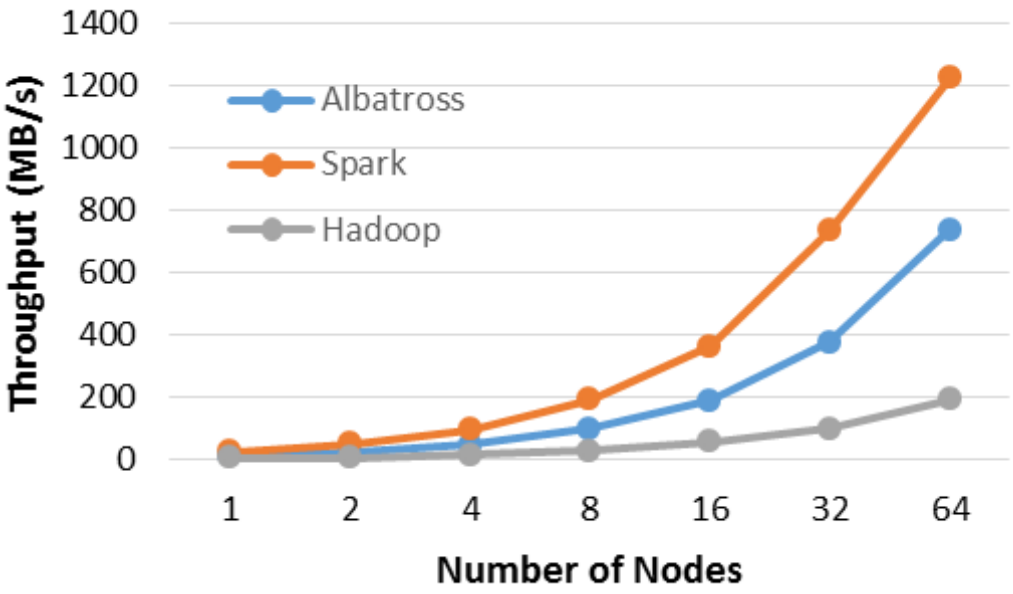

Figure 60. Word-count application Throughput

Figure 61 shows the throughput of word-count using different partition sizes on 64 nodes. Spark outperforms Albatross on 100MB partitions. However, it could not keep a steady performance at smaller partition sizes. Albatross goes from being slightly slower at the largest partition size to outperforming Spark by 3.8x. Spark is not able to schedule tasks at higher task rates. Hence the throughput drops on smaller scales.

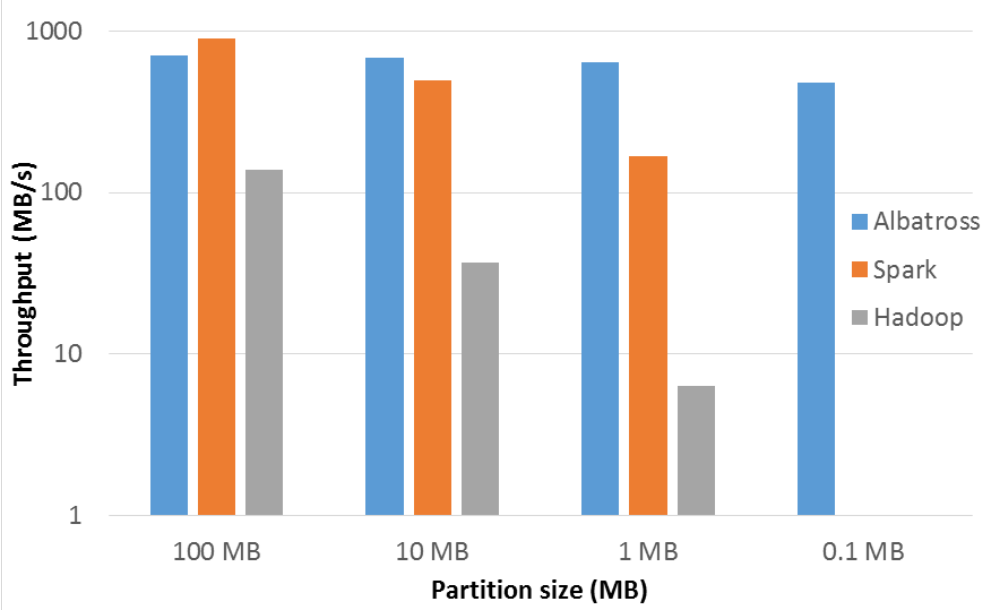

Figure 61. Word-count throughput (varied partition sizes)

Figure 62 compares the performance of grep application on the three systems using large partitions. Albatross outperforms the Spark and Hadoop by 2.13x and 12.2x respectively.

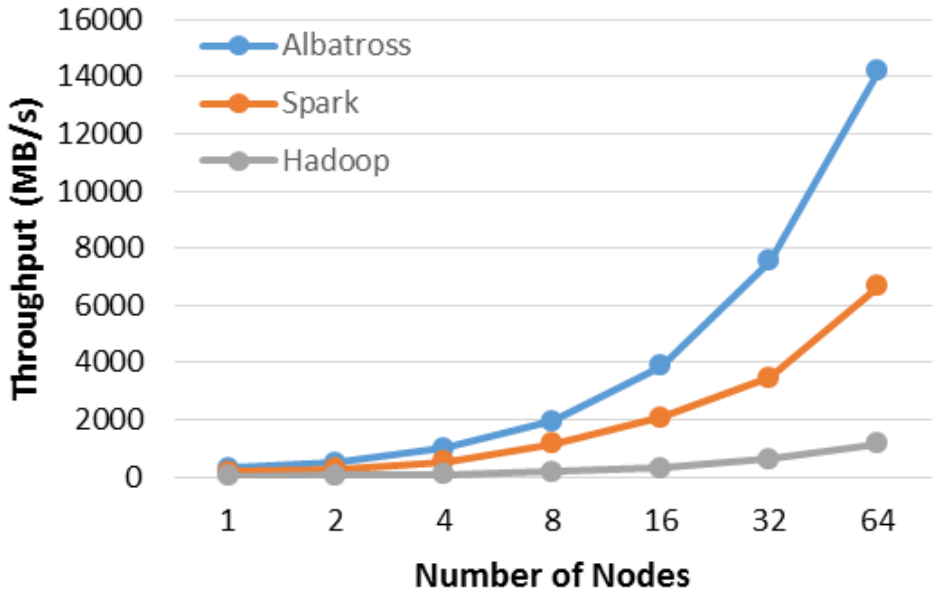

Figure 62. Throughput of grep application

Figure 63 shows the throughput of grep using different partition sizes on 64 nodes. As the partition size gets smaller, the gap between the throughput of Albatross and the other two systems becomes more significant. Similar to the other applications, the throughput of Albatross is stable on different partition sizes.

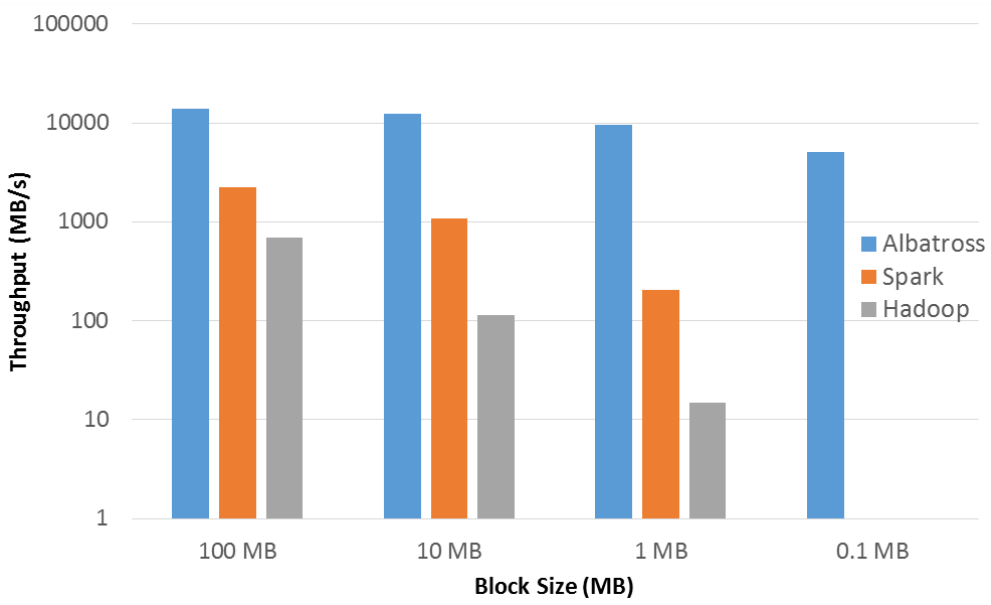

Figure 63. Grep application throughput (varied partition sizes)

5.5 Summary

Over the past few years, Data analytics frameworks such as Spark and Hadoop have gained a great deal of attraction. With the growth of Big Data and the transition of workloads and applications to high granularity tasks with shorter run time, the requirements for the data analytics has changed. Centralized frameworks will no longer be able to schedule and process the big datasets. There is need for distributed task scheduling and execution systems. This paper proposes Albatross, a distributed task scheduling and execution framework that is able to handle tasks with very high granularities. Albatross uses a task pulling approach as opposed to the traditional scheduling systems. Instead of pushing the tasks to the workers by a central or regional scheduler, in Albatross, workers pull tasks from a distributed message queue system. That leads to a scalable system that could achieve good load balancing and high

utilization. Albatross avoids any centralized components in its design. Each node could be a worker, a server, or a client at the same time. The DMQ and the DHT are distributed among all of the nodes in the system. The task submission, scheduling, and the execution are taken place through the collaboration of all of the system nodes.

Our evaluations prove that Albatross outperforms Spark and Hadoop in scheduling microbenchmarks and real applications. As can be seen, both Spark and Hadoop have centralized bottlenecks which become detrimental to their overall performance and system utilization as the task granularity decreases. Although failover options are available for both Spark and Hadoop, this will not be sufficient as even the failover node will not be able to keep up-to-date with the surplus of quick tasks waiting in its queue. Therefore, a tradeoff between task heterogeneity and performance is prevalent in these systems, but Albatross provides the user with the ability to both have heterogeneous tasks (by type and execution time) and consistently good performance. Albatross outperforms Spark and Hadoop in case of running high granular workloads with small data partitions and tasks. The task scheduling rate on Albatross is almost an order of magnitude higher than what Spark could achieve. Albatross was able to provide a high and stable throughput and latency on partition sizes as low as 100KB.

CHAPTER 6

RELATED WORK

This section introduces the related work of our proposal, which covers a wide range of research topics and areas. The related work could be divided into several aspects, namely, the evaluation of the performance of the cloud for scientific computing, distributed job scheduling systems, and the distributed message queues.

6.1 The Cloud Performance for Scientific Computing

There have been many researches that have tried to evaluate the performance of Amazon EC2 cloud [34][21][35]. However the experiments were mostly run on limited types and number of instances. Therefore they lack the generality in their results and conclusions, as they have not covered all instance types.

Ostermann et al. have evaluated Amazon EC2 using micro-benchmarks in different performance metrics. However their experiments do not include the more high-end instances that are more competitive to HPC systems. Moreover, the Amazon performance has improved since then and more features have been added to make it useful for HPC applications [34]. In addition to the experiments scope of that paper, our work provides the evaluations of the raw performance of a variety of the instances including the high-end instances, as well as the performance of the real applications.

He et al. have deployed a NASA climate prediction application into major public clouds, and compared the results with dedicated HPC systems results. They have run

micro-benchmarks and real applications [20]. How-ever they only run their experiments on small number of VMs. We have evaluated the performance of EC2 on larger scales.

Jackson has deployed a full application that performs massive file operations and data transfer on Amazon EC2 [36]. The research mostly focuses on different storage options on Amazon. Walker evaluates the performance of EC2 on NPB benchmarks and compares their performance on EC2 versus NCSA ABE supercomputer on limited scale of 1 and 4 instances [23]. The paper suffices to bring the results without detailed analysis and does not identify what this gap contributes to. Other papers have run the same benchmark on different infrastructures and provided better analysis of the results [20][50].

Only a few of the researches that measure the applicability of clouds for scientific applications have used the new Amazon EC2 cluster instances that we have tested [19][38][41]. Mehrotra compares the performances of Amazon EC2 HPC instances to that of NASA's Pleiades supercomputer [19]. However the performance metrics in that paper is very limited. They have not evaluated different performance metrics of the HPC instances. Ramakrishnan have measured the performance of the HPCC benchmarks [38]. They have also applied two real applications of PARATEC and MILC.

Juve investigates different options of data manage-ment of the workflows on EC2 [41]. The paper evaluates the runtime of different workflows with different under-lying storage options. The aforementioned works have not provided a comprehensive evaluation of the HPC instances. Their experiments are limited to a few metrics. Among the works that have looked at the new HPC instances, our work is the only one that has

evaluated all of the critical performance metrics such as memory, compute, and network performance.

Jackson compares the conventional HPC platforms to EC2 using real applications on small scales. The evaluation results show poor performance from EC2 virtual cluster running scientific applications. However they haven't used HPC instances, and have used instances with slower interconnects. Apart from the virtualization overhead, the instances are not quite comparable to highly tuned nodes on the super computers [39].

Many works have covered the performance of public clouds without having an idea about the host performance of the nodes without virtualization overhead [34][20][21]. Younge has evaluated the performance of different virtualization techniques on FutureGrid private cloud [31]. The focus of that work is on the virtualization layer rather than the cloud infrastructure. Gupta in identifies the best fit for the cloud among the HPC applications [50]. He investigates the co-existence of the cloud with super computers and suggests a hybrid infrastructure run for HPC applications that fit into the cloud environment. The paper also provides the cost analysis of running cloud on different HPC applications and shows where it is beneficial to use cloud.

Many papers have analyzed the cost of the cloud as an alternative resource to dedicated HPC resources [36][37][42]. Our work covers the storage services performance both on micro-benchmarks as well as the performance while being used by data-intensive applications.

Our work is unique in a sense that it provides comprehensive evaluation of EC2 cloud in different aspects. We first evaluate the performance of all instance types in order to

better identify their potentials and enable users to choose the best instances for different use case scenarios. After identifying the potentials, we compare the performance of the public cloud and a private cloud on different aspects, running both microbenchmarks and real scientific applications. Being able to measure the virtualization overhead on the FermiCloud as a private cloud, we could provide a more realistic evaluation of EC2 by comparing it to the FermiCloud.

Another important feature of the Cloud is having different services. We provide a broader view of EC2 by analyzing the performance of cloud services that could be used in modern scientific applications. More scientific frameworks and applications have turned into using cloud services to better utilize the potential of Cloud [37][46]. We evaluate the performance of the ser-vices such Amazon S3 and DynamoDB as well as their open source alternatives running on cloud. Finally, this work is unique in comparing the cost of different instances based on major performance factors in order to find the best use case for different instances of Amazon EC2.

6.2 Distributed Scheduling Frameworks

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 [6] was implemented to harness the unused CPU cycles on workstations for long-running batch jobs. Slurm [4] 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) [7] was originally developed to address the needs of HPC. It can manage batch and inter-active jobs, and add the ability to signal, rerun and alter jobs. LSF Batch [57] 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 [18] 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 [12]. A hierarchical implementation of Falkon was shown to scale to a petascale system in [12], 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) [56][57][58][59]. In [59], 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 [58]. Charm++ [60] supports centralized, hierarchical and distributed load balancing. In [60], 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 [14]. 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 [61]. 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 [60], to load balance threads on shared memory parallel machines [63][64][13]. 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 [13]. The largest studies to date of work stealing have been at thousands of cores scales, showing good to excellent efficiency depending on the workloads [13]. MATRIX is an execution fabric that focuses on running Many Task Computing (MTC) jobs [23]. 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++ [22] , 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.

6.3 Distributed Message Queues

Enterprise queue systems are not new in the distributed computing area. They have been around for quite a long time and have played a major role in asynchronous data movement. Systems like JMS [84] and IBM Websphere MQ [83] have been used in distributed applications. However, these systems have some limitations that make them unusable for today's big data computing systems. First, these systems usually add significant overhead to the message flow that makes them incapable of handling large scale data flows. JMS supports delivery acknowledgement for each message. IBM Websphere MQ provides atomic transaction support that lets the publisher submit a message to all of the clients. These features can add significant overhead to the process. It

is not trivial to handle these features for the larger scale systems. In general, traditional queuing services have many assumptions that prevent them from scaling well. Also, many of these traditional services do not support persistence.

ActiveMQ [82] is a message broker in Java that supports AMQP protocol. It also provides a JMS client. ActiveMQ provides many configurations and features. But it does not support any message delivery guarantee. Messages could be delivered twice or even get lost. Other researches have shown that it cannot scale very well in larger scales [76].

RabbitMQ [81] is a robust enterprise queuing system with a centralized manager. The platform provides option to choose between performance and reliability. That means enabling persistence would highly degrade the performance. Other than the persistence, the platform also provides options like delivery acknowledgement and mirroring of the servers. The message latency on RabbitMQ is large and not tolerable for any application that is sensitive to the efficiency. Being centralized makes RabbitMQ not scale very well. It also makes it unreliable because of having a single point of failure. Other researches have shown that it cannot perform well in larger scales as compared to scalable systems like Kafka [76].

Besides the traditional queuing systems, there are two modern queuing services that have got quite popular among commercial and open source user community. Those two are Apache Kafka and Amazon Simple Queue Service (SQS) [16]. Kafka is an open source, distributed publish and consume service which is introduced by LinkedIn. The design goal of Kafka is to provide a system that gathers the logs from a large number of servers, and feeds it into HDFS [85] or other analysis clusters. Other log management

systems that were provided by other big companies are usually saving data to offline file systems and data warehouses. That means they do not have to provide low latency. However, Kafka can deliver data to both offline and online systems. Therefore, it needs to provide low latency on message delivery. Kafka is fully distributed and provides high throughput. We discuss more about Kafka later in a separate section.

Amazon SQS is a well-known commercial service which provides reliable message delivery in large scales. SQS is persistent. Like many other Amazon AWS services [15], SQS is reliable and highly available. It is fully distributed and highly scalable. We discuss more about SQS and compare its features to Fabriq in another section.

6.4 Distributed Job Scheduling and Execution Frameworks

There have been many works providing solutions for task or job scheduling in distributed resources. Some of those works have focused on task scheduling and execution while some have focused on resource management on large clusters. Frameworks such as Condor, Slurm and PBS were mainly designed for batch-jobs that take longer to run. They are mainly used as the main resource manager and job submission framework of the large clusters. The main limitation of those frameworks is their centralized architecture that makes them not capable of handling larger scales. They were all designed for longer running batch jobs and are unable to schedule workloads in fine granular task level.

Systems such as Mesos [113], and Omega are resource managers that were designed for allocating resources to different applications and users that share a large cluster. Another framework with similar functionality is Borg [122]. Borg is a cluster manager

designed and implemented by Google. Borg focuses on optimizing the resource utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. All of these frameworks provide resources to the user applications and jobs. There is no concept of task level scheduling or execution framework that could natively run the workloads. More importantly, these systems have centralized architectures and could not be the ultimate solution for distributed processing of data analytics workloads. The nature of Albatross is different from those systems as Albatross provides a native execution framework. With Albatross, the user only needs to provide the driver program and everything else including the data distribution, locality and the task execution dependency is handled by the framework.

Many industrial systems such as Spark & Hadoop utilize iterative transformations and the Map-reduce model, respectively, but still exhibit bottlenecks, particularly the centralized task/resource managers [30].

Hadoop is a data analytics framework that adopted its architecture from Google's Map-reduce implementation. It consists of two main components which are the distributed file system (HDFS) and the Map-reduce programming paradigm. The two main centralized components of Hadoop are the NameNode and the JobTracker. The JobTracker is in charge of tracking any disk reads/writes to HDFS. As the job tracker must be notified by the task trackers, similarly the NameNode must be notified of block updates executed by the data nodes.

Spark, like many other state-of-the-art systems, utilizes a master-slave architecture. The flexible transformations enable Spark to manipulate and process a workload in a more efficient manner than Hadoop. Spark consists of a centralized job-level scheduler for any jobs submitted to the cluster. Similar to Hadoop, centralized bottlenecks are still present even though the capability of iterative workloads is better handled in Spark than in Hadoop.

Usually a centralized version is relatively simple to implement. However, as seen in the performance evaluation in previous chapter, these centralized components may be the downfall of the system as scale increases. Additionally, systems such as Sparrow [8], which try to reduce the consequences associated with Spark's centralized job scheduler. Sparrow has a decentralized architecture that makes it scalable. Although it provides a distributed scheduler for the jobs, the task-level scheduler is still centralized and therefore, provides no improvement in our experiments. Sparrow sidesteps the bottleneck of Spark by providing each job, a separate allocated scheduler. Therefore, the task scheduling on the system remains centralized. Even though that solves the problem to a certain level, it still could not solve the problem of running a single job with a large amount of highly granulated short-length tasks.

Apache Flink is an in-memory distributed big data processing platform with goals similar to Hadoop and Spark. Flink was mainly designed to optimize the processing of distributed batch data and stream data. The main component of Flink is the streaming dataflow engine which is responsible for data distribution, communication, and fault tolerance for distributed computations over data streams. Flink's execution engine treats

batch data as a data stream with a finite dataset and time. The major novelty of Flink is using pipelines on its design that makes it a perfect fit for stream data. Using pipelines and iteration operators enables Flink to provide native support for iterative applications. Moreover, due to using pipelines, Flink is able to overlap between different stages.

Flink has been able to outperform frameworks like Spark and Hadoop due to its optimized architecture for streaming and batch data. On iterative workloads, Spark users need to generate an extra set of tasks for each iteration. However, in Flink there is no need for an explicit set of tasks for each iteration and the execution engine is able to natively run the multiple iterations until they meet a certain criteria defined by the user.

Flink has many different modes for its job distribution. User can use Yarn, the Flink's default stand-alone scheduler, or Mesos for workload distribution. The main limitation of the Flink is the centralized architecture of its job distributor. All of the above mentioned options schedule and distribute jobs from centralized components. As discussed before on previous sections, the centralized architecture limits the scalability of the system. Moreover, due to using separate and independent pipelines Flink does not have a job level fault tolerance support. That means if a tasks fails, the whole job needs to be restarted.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

We draw the conclusions achieved and highlight the future work towards developing next-generation job management system for extreme-scale computing.

7.1 Conclusions

According to our achievements and the findings on our papers, we are able to make the following conclusions.

Cloud Computing has potentials to run Scientific Applications: We showed that certain series of compute instances and other type of resources have the ability to perform closely to the HPC clusters and Supercomputers. Therefore there is a potential to provide tools, such as scheduling systems to run scientific workloads and applications.

We present a comprehensive, quantitative study to evaluate the performance of the Amazon EC2 for the goal of running scientific applications. Our study first covers the performance evaluation of different compute instances in terms of compute, network, memory and storage. In terms of compute performance, only a few of the instances that are from a certain group were able to meet the requirements of the scientific workloads. We also noticed that there is need for improvements on network performance. Moreover, we compared the performance of the real distributed applications on the cloud and observed that their performance is close to each other. We also have to state that the new generation of the AWS instances which were released after this research was conducted can possibly satisfy the needs of the scientific and HPC applications [112].

Distributed Message Queues can be used as building blocks for distributed scheduling: We designed and implemented CloudKon, a distributed job scheduling system using SQS as its building block. We assessed the throughput, latency and the scalability of SQS to make sure that it could be used as a building block for our scheduling system. Using SQS gave us the ability to design a powerful and scalable scheduling system with loosely coupled components. We were able to prove that it is a good idea to use cloud services within distributed applications that run inside the cloud. It is not trivial to create complex distributed applications that could perform well on cloud. Using cloud services will make it significantly easier and faster to implement these applications and guarantees the optimal performance and utilization. We were able to support both HPC and MTC workloads on CloudKon by using SQS.

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.

Distributed Hash Tables are building blocks for large scale system services: We motivated that Distributed Hash Tables (DHT) are a viable building block for large scale distributed applications and tools. This statement lays the foundations for developing distributed system services that are highly available, scalable and reliable at larger scales.

We implemented Fabriq, a distributed message queue that runs on top of ZHT. ZHT is a scalable DHT that is proven to perform well at larger scales and achieve sub-milliseconds latency. It is also reliable and persistent. That makes it a perfect match to be used as a building block of a distributed application. The design goal of Fabriq was to achieve low latency and high throughput while maintaining the perfect load balance among its nodes at larger scales. We used ZHT as a building block for Fabriq.

The results show that Fabriq outperforms Kafka and SQS in different metrics. Fabriq has shown to have low overhead on the data movement process. Thus it achieves a higher efficiency than the other two systems. It also has a faster message delivery. Message delivery latency on SQS and Kafka is orders of magnitude more than Fabriq. Moreover, they have a long range of push and pop latency which makes them unsuitable for applications that are sensitive to operations with long tails. Fabriq provides a very stable latency throughout the delivery. Results show that more than 90% of the operations take less than 0.9ms and more than 99% percent of the operations take less than 8.3ms in Fabriq. Fabriq also achieves high throughput is large scales for both small and large messages. At the scale of 128, Fabriq was able to achieve more than 90000 msgs/sec for small messages. At the same scale, Fabriq was able to deliver large messages at the speed of 1.8 GB/sec.

Next generation scheduling is moving to the worker nodes collaboration: In order to achieve resource management at the extreme-scales, the scheduling, and the control should move from the traditional approach of scheduling via central controllers to scheduling through the collaboration of all of the workers. We designed and implemented

Albatross, a task level scheduling and execution framework that uses a Distributed Message Queue (DMQ) for task distribution among its workers. The key insight behind the Albatross is that unlike other conventional schedulers, in Albatross, a worker is the active controller and the decision making component of the framework. Hence, no component in Albatross will be a bottleneck for increasing the scale. Moreover, using DMQs, Albatross was able to provide native support for task execution dependency and data locality which are both necessary for the modern data analytics workloads. Compared to the traditional approaches, such an approach could tolerate higher rates of scheduling for very high granular tasks. Albatross outperforms Spark and Hadoop in case of running high granular workloads with small data partitions and tasks. The task scheduling rate on Albatross is almost an order of magnitude higher than what Spark could achieve. Albatross is able to schedule tasks at 10K tasks per second rate, outperforming Spark by 10x. The latency of Albatross is almost an order of magnitude lower than Spark. Albatross's throughput on real applications has been faster than the two other systems by 2.1x and 12.2x. Finally, it outperforms Spark and Hadoop respectively by 46x, and 600x in processing high granularity workloads on grep application.

7.3 Future Work

The achievements we have accomplished to date have laid the foundations for a broad yet clear sort of future work. We were able to: (1) Assess the ability of the cloud for running scientific applications. (2) Design and implement a distributed job scheduling system that runs on Amazon AWS using SQS. (3) Design and implement a distributed and scalable message queue service that could potentially replace SQS. (4) Design and implement task level scheduling and execution framework that is able to run synthetic and data analytics applications at large scales.

All the future work aims to push our resource management system to scalable production systems that could run a variety of applications on the cloud environment efficiently. The future directions of our work are listed as follows:

Native support for complex data processing workloads: Distributed task scheduling and execution frameworks such as Spark or Hadoop have been heavily used to solve data processing problems. Data processing workloads could adapt well to the Map-Reduce paradigm of the above mentioned frameworks. However, some data process workloads have a more complex process than the simple two step logic of Map-Reduce paradigm. Those workloads are usually very hard to implement, if not impossible to implement with frameworks like Hadoop. Albatross has the ability to natively execute such workloads. However, in Albatross each iteration is treated as a new set of tasks. Flink, which is another distributed data processing platform deal with iterative applications in a more enhanced way. A series of iterations with a certain duty would only generate a single task. In the future work, we would like to adapt that feature that could optimize the efficiency

of Albatross while running iteration applications. We are also going to run a set of iterative applications that have longer and more complex DAGs. Having a set of features such as the execution dependency control and data locality enables Albatross to run the users schema in a single run.

Native support for interactive data analysis workloads: Another future direction of this work is to enable support for interactive data analysis and interactive queries on massive scales of data. Currently some of the scheduling and execution frameworks support interactive jobs. For example, Dremel [12] supports interactive queries. However it has a hierarchical architecture that makes it unable to scale very well. Being able to support more complex workflows, Albatross should be able to perform well while executing such workloads.

Optimum Load Balancing on different Scenarios: As we stated on the section 4.3 of chapter 4, Fabriq provides near optimum load balancing among its servers while distributing and delivering the messages. It achieves the perfect load balancing by making use of the static hashing over its servers. However, that is only true when Fabriq has a static set of servers running from the beginning of its bootstrap. Fabriq supports dynamic membership. That means it is possible to add or remove servers in the middle of the process without crashing the system. The perfect load balancing is not guaranteed anymore if one makes use of the dynamic membership. That means if a server is added or removed from the Fabriq dynamically, it will not be able to achieve a perfect load balance any more. One of the future directions of Fabriq is to provide perfect load balance under this corner case situation.

Distributed Monitoring Via Distributed Message Queues: Monitoring has proven to be essential for distributed systems. It is very important to understand how the resources of a distributed system are utilized by applications. Monitoring resources of a large scale distributed system is not trivial with a centralized traditional monitoring solution. Distributed Message Queues could play an essential part to provide a distributed solution for large scale system monitoring [101]. One of the future directions of this work is to design and implement a distributed monitoring system using Fabriq. Providing an efficient system, Fabriq will be able to serve a distributed monitoring solution well.

BIBLIOGRAPHY

[1] M. Wall, “Big Data: Are you ready for blast-off”, BBC Business News, March 2014

[2] I. Raicu, P. Beckman, I. Foster. “Making a Case for Distributed File Systems at Exascale”, Invited Paper, LSAP, 2011

[3] V. Sarkar, S. Amarasinghe, et al. “ExaScale Software Study: Software Challenges in Extreme Scale Systems”, ExaScale Computing Study, DARPA IPTO, 2009.

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

[5] 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.

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

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

[8] 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.

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

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

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

[12] 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

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

[14] 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.

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

[16] Amazon SQS, [online] 2014, http://aws.amazon.com/sqs/

[17] W. Voegels. “Amazon DynamoDB, a fast and scalable NoSQL database service designed for Internet-scale applications.”

http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html, January 18, 2012

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

[19] 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.

[20] 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.

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

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

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

[24] Amazon EC2 Instance Types, Amazon Web Services, [online] 2013, http://aws.amazon.com/ec2/instance-types/ (Accessed: 2 November 2013)

[25] Amazon Simple Storage Service (Amazon S3), Amazon Web Services, [online] 2013, http://aws.amazon.com/s3/ (Accessed: 2 November 2013)

[26] Iperf, Souceforge, [online] June 2011, http://sourceforge.net/projects/iperf/ (Accessed: 2 November 2013)

[27] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary. "HPL", (netlib.org), [online] September 2008, http://www.netlib.org/benchmark/hpl/ (Accessed: 2 November 2013)

[28] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the MPI standard," Tech. Rep. CS-95-274, University of Tennessee, Jan. 1995

[29] Release: Amazon EC2 on 2007-07-12, Amazon Web Services, [online] 2013, http://aws.amazon.com/releasenotes/Amazon-EC2/3964 (Accessed: 1 November 2013)

[30] K. Yelick, S. Coghlan, B. Draney, and R. S. Canon, "The Magellan report on cloud computing for science," U.S. Department of Energy, Washington DC, USA, Tech. Rep., 2011. Available: http://www.nersc.gov/assets/StaffPublications/2012/MagellanFinalReport.pdf

[31] A. J. Younge, R. Henschel, J. T. Brown, G. von Laszewski, J. Qiu, and G. C. Fox, "Analysis of virtualization technologies for high performance computing environments," International Conference on Cloud Computing, 2011

[32] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. "Swift: Fast, reliable, loosely coupled parallel computation", IEEE Int. Workshop on Scientific Workflows, pages 199–206, 2007

[33] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, B. Clifford. "Towards Loosely-Coupled Programming on Petascale Systems", IEEE/ACM Supercomputing, pages 1-12, 2008

[34] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. "A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing". In Cloudcomp, 2009

[35] S. L. Garfinkel, "An evaluation of amazon's grid computing services: EC2, S3 and SQS," Computer Science Group, Harvard University, Technical Report, 2007, tR-08-07

[36] K. R. Jackson, K. Muriki, L. Ramakrishnan, K. J. Runge, and R. C. Thomas. "Performance and cost analysis of the supernova factory on the amazon aws cloud". Scientific Programming, 19(2-3):107-119, 2011

[37] J.-S. Vockler, G. Juve, E. Deelman, M. Rynge, and G.B. Berriman, "Experiences Using Cloud Computing for A Scientific Workflow Application," 2nd Workshop on Scientific Cloud Computing (ScienceCloud), 2011

[38] L. Ramakrishnan, P. T. Zbiegel, S. Campbell, R. Bradshaw, R. S. Canon, S. Coghlan, I. Sakrejda, N. Desai, T. Declerck, and A. Liu. "Magellan: experiences from a science cloud". In Proceedings of the 2nd international workshop on Scientific cloud computing, pages 49–58, San Jose, USA, 2011

[39] K. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. Wasserman, and N. Wright, "Performance Analysis of High Performance Computing

Applications on the Amazon Web Services Cloud," in 2nd IEEE International Conference on Cloud Computing Technology and Science. IEEE, 2010, pp. 159–168

[40] J. Lange, K. Pedretti, P. Dinda, P. Bridges, C. Bae, P. Soltero, A. Merritt, "Minimal Overhead Virtualization of a Large Scale Supercomputer," In Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011), 2011.

[41] G. Juve, E. Deelman, G.B. Berriman, B.P. Berman, and P. Maechling, 2012, "An Evaluation of the Cost and Performance of Scientific Workflows on Amazon EC2", Journal of Grid Computing, v. 10, n. 1 (mar.), p. 5–21

[42] R. Fourer, D. M. Gay, and B. Kernighan, "AMPL: a mathematical programming language" in Algorithms and model formulations in mathematical programming, 1st Ed. New York, NY: Springer-Verlag New York, Inc., 1989, ch., pp. 150–151.

[43] FermiCloud, Fermilab Computing Sector, [online], http://fclweb.fnal.gov/ (Accessed: 25 April 2014)

[44] R.Moreno-vozmendiano, S. Montero, I. Llorente." IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures: Digital Forensics", Computer (Long Beach, CA)

[45] K. Hwang, J. Dongarra, and G. C. Fox," Distributed and Cloud Computing: From Parallel Processing to the Internet of Things". Morgan Kaufmann, 2011

[46] I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Ramamurty, K. Wang, and I. Raicu. "Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing." In Proc. 14th

IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'14). 2014

[47] T. L. Ruivo, G. B. Altayo, G. Garzoglio, S. Timm, K. Hyun, N. Seo-Young, and I. Raicu. "Exploring Infiniband Hardware Virtualization in OpenNebula towards Efficient High-Performance Computing." CCGrid'14. 2014

[48] D. Macleod, (2013). OpenNebula KVM SR-IOV driver.

[49] K. Maheshwari, K. Birman, J. Wozniak, and D.V. Zandt. "Evaluating Cloud Computing Techniques for Smart Power Grid Design using Parallel Scripting" CCGrid'13, pages 319-326, 2013

[50] A. Gupta, L. Kale, F. Gioachin, V. March, C. Suen, P. Faraboschi, R. Kaufmann, and D. Milojicic. "The Who, What, Why and How of High Performance Computing Applications in the Cloud," HP Labs, Tech. Rep. HPL-2013-49, July 2013.

[51] E. Walker, "Benchmarking amazon ec2 for high-performance scientific computing," in USENIX; login: magzine, Oct. 2008.

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

[53] 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.

[54] 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

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

[56] 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.

[57] 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.

[58] 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.

[59] 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

[60] 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.

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

[62] M. 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.

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

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

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

[66] 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.

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

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

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

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

[71] 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

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

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

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

[75] W. Vogels. “Improving the Cloud - More Efficient Queuing with SQS” [online] 2012, http://www.allthingsdistributed.com/2012/11/efficient-queueing-sqs.html

[76] J. Kreps, N. Narkhede, and J. Rao. “Kafka: A distributed messaging system for log processing”. NetDB, 2011.

[77] A. Alten-Lorenz, Apache Flume, [online] 2013, https://cwiki.apache.org/FLUME/

[78] A. Thusoo, Z. Shao, et al, “Data warehousing and analytics infrastructure at facebook,” in SIGMOD Conference, 2010, pp. 1013–1020.

[79] J. Pearce, Scribe, [online] https://github.com/facebookarchive/scribe

[80] T. J. Hacker and Z. Meglicki, “Using queue structures to improve job reliability,” in Proceedings of the 16th International Symposium on High-Performance Distributed Computing (HPDC), 2007, pp. 43–54.

[81] A. Videla and J. J. Williams, “RabbitMQ in action”. Manning, 2012.

[82] B. Snyder, D. Bosanac, And R. Davies, “ActiveMQ in action” Manning, 2011.

[83] S. Davies, and P. Broadhurst, “WebSphere MQ V6 Fundamentals”, IBM Redbooks, 2005

[84] Java Message Service Concepts, Oracle, [online] 2013, http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

[85] D.Borthakur, HDFS architecture. Tech. rep., Apache Software Foundation, 2008.

[86] P. Hunt, et. al., “ZooKeeper: wait-free coordination for internet-scale systems,” Proceedings of USENIXATC’10, 2010.

[87] J. Lin, D. Ryaboy. “Scaling big data mining infrastructure: The twitter experience.” SIGKDD Explorations, 14(2), 2012.

[88] A. Reda, Y. Park, M. Tiwari, C. Posse, S. Shah. “Metaphor: a system for related search recommendations.” In CIKM, 2012.

[89] R. Ramesh, L. Hu, and K. Schwan. “Project Hoover: auto-scaling streaming map-reduce applications”. Proceedings of (MBDS '12). ACM, New York, NY, USA, 7-12. 2012

[90] H. Liu, “Cutting map-reduce cost with spot market”. 3rd USENIX Workshop on Hot Topics in Cloud Computing (2011).

[91] T. White, “Hadoop: The Definitive Guide.” O’Reilly Media, Inc., 2009

[92] K. Wang, et. al. "Optimizing Load Balancing and Data-Locality with Data-aware Scheduling", IEEE Big Data 2014.

[93] I. Kapur, K. Belgodu, P. Purandare, I. Sadooghi, I. Raicu. "Extending CloudKon to Support HPC Job Scheduling", Illinois Institute of Technology, Department of Computer Science, Technical Report, 2013

[94] A. Anthony, S. Palur, I. Sadooghi, I. Raicu. "CloudKon Reloaded with efficient Monitoring, Bundled Response and Dynamic Provisioning", Illinois Institute of Technology, Department of Computer Science, Technical Report, 2013

[95] D. Patel, F. Khasib, S. Srivastava, I. Sadooghi, I. Raicu. "HDMQ: Towards In-Order and Exactly-Once Delivery using Hierarchical Distributed Message Queues", Illinois Institute of Technology, Department of Computer Science, Technical Report, 2013

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

[97] D. Zhao, C. Shou, Z. Zhang, I. Sadooghi, X. Zhou, T. Li, I. Raicu. "FusionFS: a distributed file system for large scale data-intensive computing", 2nd Greater Chicago Area System Research Workshop (GCASR), 2013.

[98] I. Sadooghi, I. Raicu. "Understanding the Cost of the Cloud for Scientific Applications", 2nd Greater Chicago Area System Research Workshop (GCASR), 2013.

[99] I. Sadooghi, D. Zhao, T. Li, I. Raicu. “Understanding the Cost of Cloud Computing and Storage”, 1st Greater Chicago Area System Research Workshop, 2012 (poster)

[100] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.” In NSDI, 2012

[101] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, “Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications,” in Proc. IEEE/ACM International Conference for High Performance Storage, Networking, and Analysis (SC14), 2014.

[102] I. Sadooghi, J. Hernandez Martin, T. Li, K. Brandstatter, Y. Zhao, K. Maheshwari, T. Pais Pitta de Lacerda Ruivo, S. Timm, G. Garzoglio, I. Raicu. “Understanding the performance and potential of cloud computing for scientific applications” IEEE Transactions on Cloud Computing PP (99), 2015.

[103] I. Sadooghi, K. Wang, S. Srivastava, D. Patel, D. Zhao, T. Li, I. Raicu. “FaBRiQ: Leveraging Distributed Hash Tables towards Distributed Publish-Subscribe Message Queues “, 2nd IEEE/ACM International Symposium on Big Data Computing (BDC) 2015.

[104] D. Zhao, I. Raicu. “Distributed File Systems for Exascale Computing”, Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012

[105] C. Dumitrescu, I. Raicu, I. Foster. "The Design, Usage, and Performance of GRUBER: A Grid uSLA-based Brokering Infrastructure", International Journal of Grid Computing, 2007

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

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

[108] D. Zhao, N. Liu, D. Kimpe, R. Ross, X. Sun, and I. Raicu. "Towards Exploring Data-Intensive Scientific Applications at Extreme Scales through Systems and Simulations", IEEE Transaction on Parallel and Distributed Systems (TPDS) Journal 2015

[109] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu. "FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems", IEEE International Conference on Big Data 2014

[110] K. Wang, A. Rajendran, K. Brandstatter, Z. Zhang, I. Raicu. “Paving the Road to Exascale with Many-Task Computing”, Doctoral Showcase, IEEE/ACM Supercomputing/SC 2012

[111] K. Wang, X. Zhou, H. Chen, M. Lang, I. Raicu. “Next Generation Job Management Systems for Extreme Scale Ensemble Computing”, ACM HPDC 2014

[112] Next Generation Cluster Computing on Amazon EC2, Amazon Web Services, [online] 2015, https://aws.amazon.com/blogs/aws/next-generation-cluster-computing-on-amazon-ec2-the-cc2-instance-type

[113] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, et. al. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In Proceedings of, NSDI'11, USENIX Association, pp. 295–308.

[114] T. Li, et. al. "A Convergence of Distributed Key-Value Storage in Cloud Computing and Supercomputing", Journal of Concurrency and Computation Practice and Experience (CCPE) 2015.

[115] K. Brandstatter, T. Li, X. Zhou, I. Raicu, Novoht: a lightweight dynamic persistent NoSQL key/value store. In: GCASR' 13, pp. 27–28, Chicago, IL (2013)

[116] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks", Euro. Conf. on Computer Systems (EuroSys), 2007.

[117] V.K. Vavilapalli, "Apache Hadoop Yarn – Resource Manager," http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/

[118] M. Noll. (2011, April) Benchmarking and Stress Testing an Hadoop Cluster With TeraSort, TestDFSIO & Co. [Online].

[119] K. Wang, K. Qiao, I. Sadooghi, et. al. "Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales", Journal of Concurrency and Computation Practice and Experience (CCPE) 2015.

[120] T. Li, I. Raicu, L. Ramakrishnan, “Scalable State Management for Scientific Applications in the Cloud”, BigData 2014/

[121] K.H. Kim, R. Buyya, J. Kim, Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled clusters, CCGrid 2007, Rio de Janeiro, Brazil, May 2007.

[122] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015, 2015, p. 18.

[123] Google Protocol Buffers: Google’s Data Interchange Format [online]. https://developers.google.com/protocol-buffers/

[124] Apache Flink, [Online] https://flink.apache.org/

[125] S. EWEN, K. TZOUMAS, M. KAUFMANN, and V. MARKL. “Spinning fast iterative data flows”. In proceedings of the VLDB 5, 11 (July 2012), 1268–1279

[126] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M.J. Sax, S. Schelter, M. Höger, K. Tzoumas, D. Warneke, “The stratosphere platform for big data analytics”, In proceedings of the VLDB J. 23 (6) (2014) 939–964

[127] D. Kim, “A Comparative Performance Evaluation of Flink” October 2015 [Online]