Albatross: an Efficient Cloud-enabled Task Scheduling and Execution Framework using Distributed Message Queues

Iman Sadooghi, Geet Kumar, Ke Wang, Dongfang Zhao, Tonglin Li, Ioan Raicu
isadoogh@iit.edu, {gkumar7, kwang22, dzhao8, tli13}@hawk.iit.edu, iraicu@cs.iit.edu
Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA

***Abstract*— 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.**

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 good 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. Our evaluations show that Albatross outperforms Spark and Hadoop at larger scales and in the case of running higher granularity workloads.

Keywords—Data Analytics, Task Scheduling, Distributed Systems, Spark, Hadoop, Distributed Task Execution, Distributed Message Queue

I. INTRODUCTION

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 scales of Petabytes or larger. 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 [2], Condor [3], PBS [4], and SGE [5]. 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 [7] and Spark [6] 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 [28]. 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 [8] 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 [27]. Our evaluations have shown that in some cases, identical instances on AWS [26] 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 [29]. 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 [9] 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) [10] 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 [11] 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 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.

The rest of the paper is organized as follows. Section II discusses the related work. Section III provides background about the two building block components of Albatross. Section IV discusses the architecture of Albatross, followed up by the implementation details of the data locality and the task execution dependency support. Next, in section V, we discuss the Map-reduce programming model support in Albatross. Section VI briefly compares the main differences of Albatross with Spark and Hadoop as the mainstream data analytics frameworks. Section VII evaluates the performance of the Albatross in different metrics. Finally, section VIII concludes the paper and discusses the future work.

II. RELATED WORK

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. Condor [3] tries to harness the unused CPU cycles on servers for batch-jobs that take longer to run. Slurm [2] is a resource

manager for Linux clusters that also provides a framework for work execution and monitoring. Portable Batch System (PBS) [4] mainly focuses on HPC. It manages batch and interactive jobs.

The main limitation of the above mentioned works 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 [12], and Omega [13] are resource managers that were designed for allocating resources to different applications. The focus of this work is on a task scheduling and execution solution that could run different types of workloads. Nevertheless, both of these systems have centralized architectures and could not be the ultimate solution for distributed processing of data analytics workloads.

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]. Usually a centralized version is relatively simple to implement. However, as seen in the performance evaluation in a later section, these centralized components may be the downfall of the system as scale increases. Additionally, there are systems such as Sparrow [8], which try to reduce the consequences associated with Spark's centralized job scheduler. It has a decentralized architecture that makes it scalable. Although Sparrow provides a distributed scheduler for the jobs, the task-level scheduler is still centralized. Therefore since we are focusing on the task level scheduling and not exploring the multiple-job workloads, Sparrow would not provide any improvement in our experiments.

Another approach that has been used for distributed scheduling is work stealing. It is used at small scales successfully in parallel languages such as Cilk [14], to load balance threads on shared memory parallel machines [15][16][17]. Scalability of work stealing has not been proven yet. The randomized nature of it could cause poor utilization and scalability [18].

III. BACKGROUND

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.

A. 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 [19].

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 [20].

B. 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. On our previous work [22], since CloudKon was using SQS [21], 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.

IV. 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.

A. 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 1. Albatross Components 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.

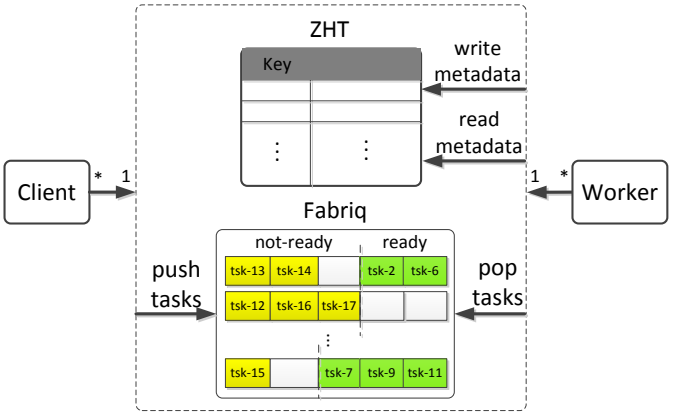

Figure 1. Albatross Components

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.

B. 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 [23], 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 2 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 the 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.

C. 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.

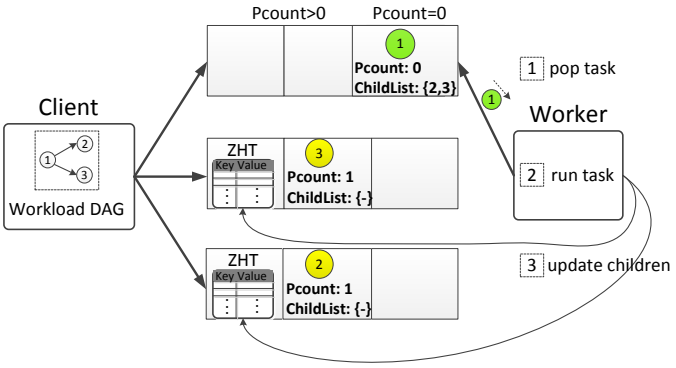

Figure 2. Task Execution Dependency Support

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 logic 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 3 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.

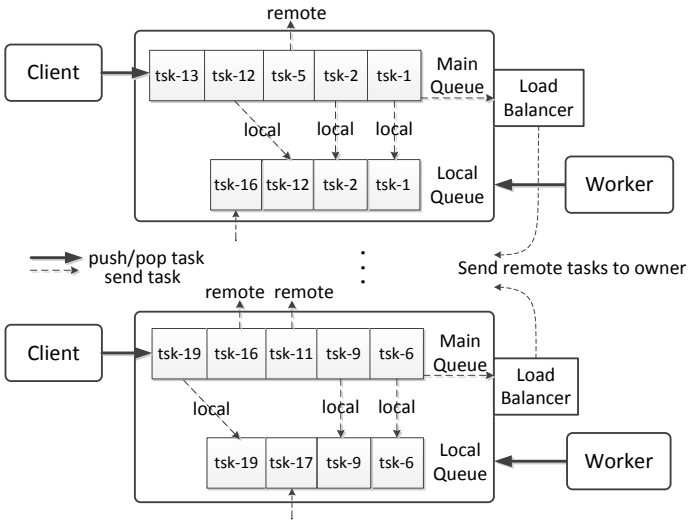

Figure 3. Data Locality Support Components

Figure 4 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 [8].

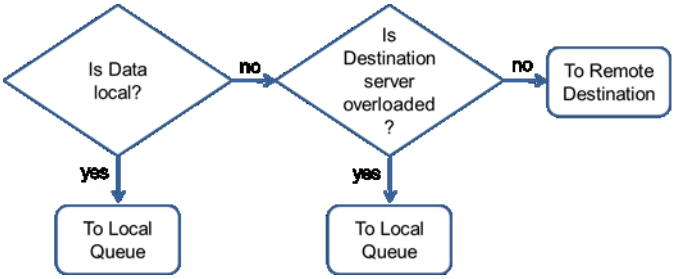

Figure 4. Locality Engine's Decision Making

V. MAP-REDUCE ON ALBATROSS

This section discusses the implementation of the Map-reduce programming model in Albatross. Figure 5 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.

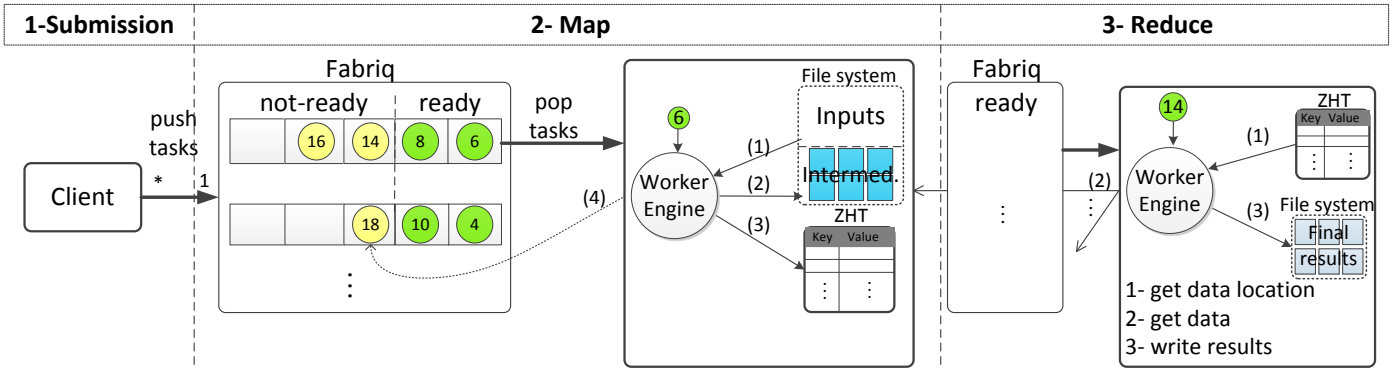

Figure 5. Map-reduce process in Albatross

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.

VI. 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.

A. Hadoop and Spark

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 [25]. Although the Yarn [24] 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.

B. 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, since the runtimes for Hadoop and Spark were very long for very short blocks, the workload was chosen to be 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.

C. *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 1,000 null tasks per node 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 6 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.

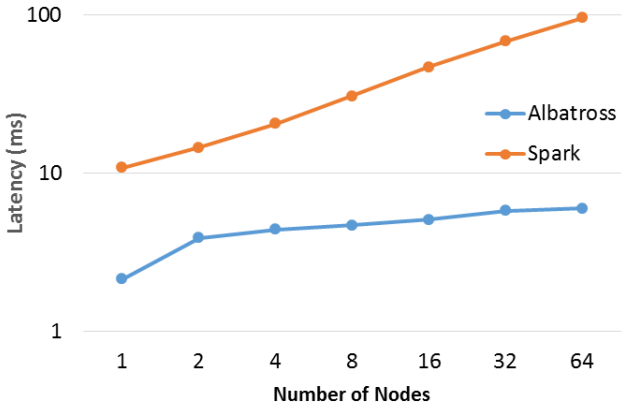

Figure 6. Average latency of in-memory null tasks

In order to fully reflect the scheduling overhead of three frameworks, we need to measure metrics like minimum, median, and maximum latency. Figure 7 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.

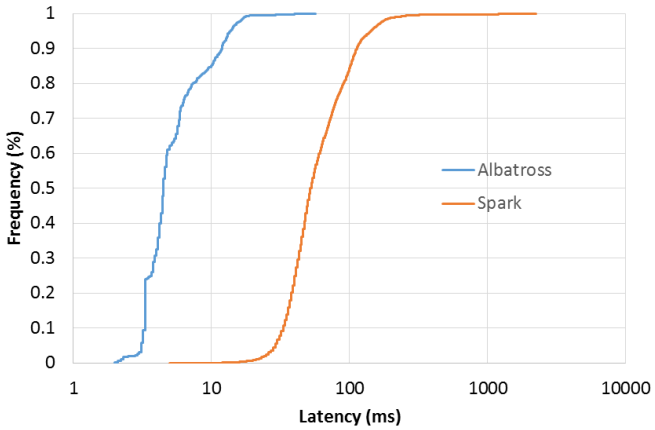

Figure 7. Cumulative distribution null tasks latency

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 8 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.

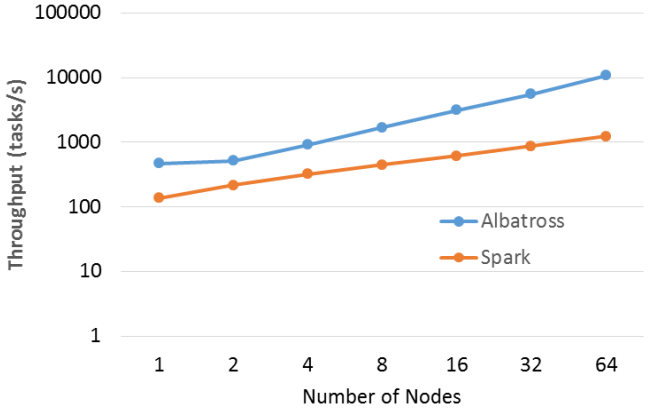

Figure 8. Throughput of null task scheduling

D. *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 [25]. Unlike the other two applications, intermediate data in sort is large and will not always fit in memory [31]. 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 9, 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.

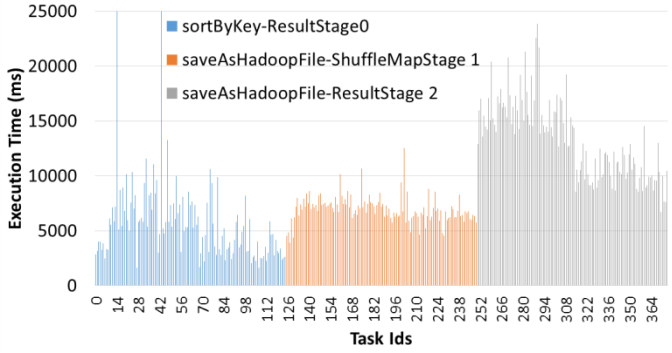

Figure 9. Spark's task breakdown (32 nodes)

As shown in Figure 10, 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.

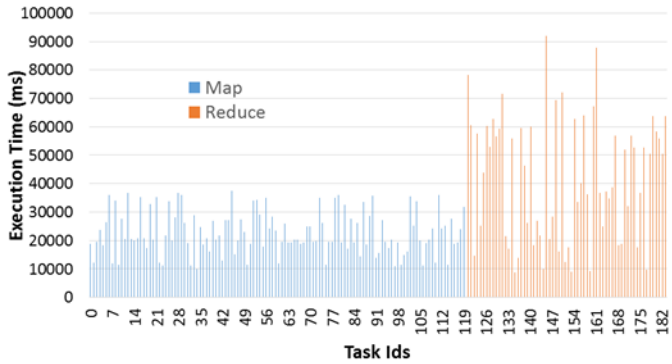

Figure 10. Hadoop's task breakdown (32 nodes)

Figure 11 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.

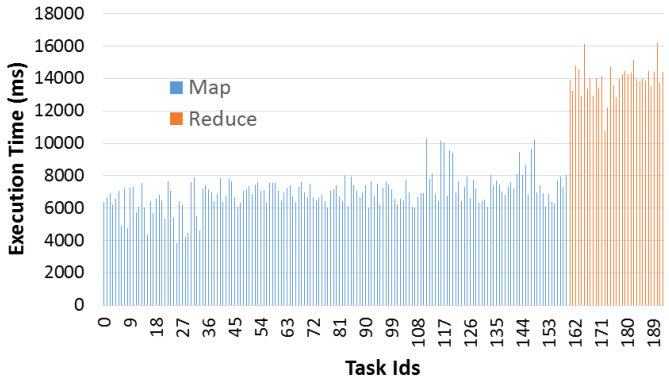

Figure 11. Task breakdown for Albatross (32 nodes)

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 12 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.

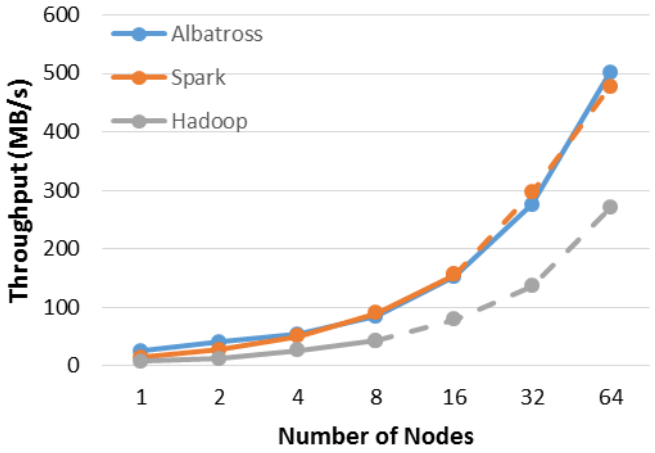

Figure 12. Throughput of sort application

Figure 13 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.

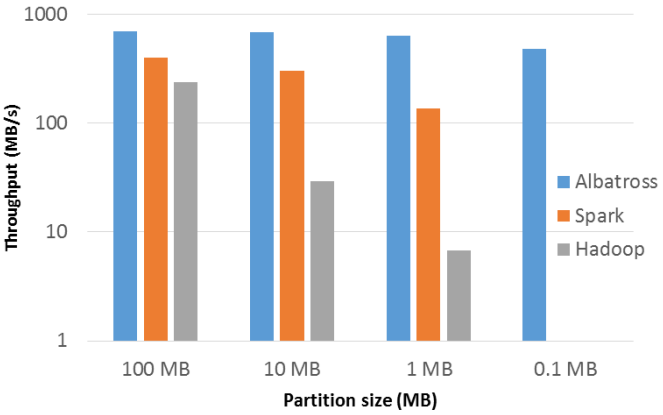

Figure 13. Sort application throughput (varied partition sizes)

Figure 14 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.

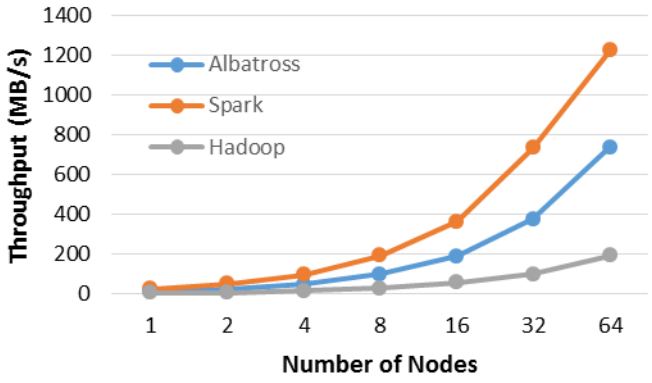

Figure 14. Word-count application Throughput

Figure 15 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.

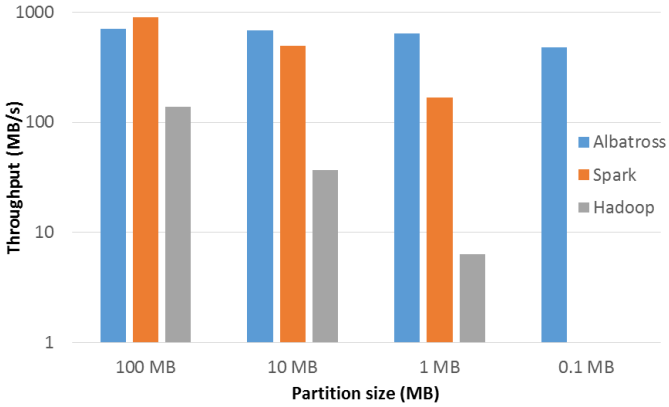

Figure 15. Word-count throughput (varied partition sizes)

Figure 16 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.

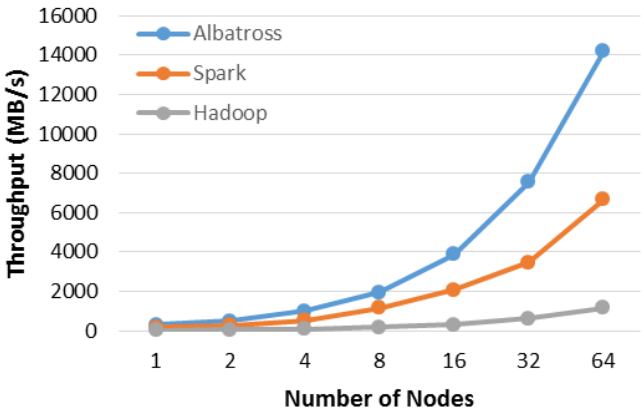

Figure 16. Throughput of grep application

Figure 17 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.

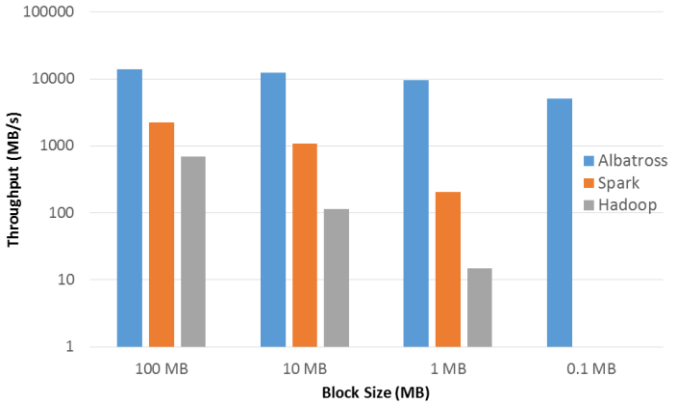

Figure 17. Grep application throughput (varied partition sizes)

VII. Conclusion and Future Work

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.

References

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

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

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

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

[5] 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.

[6] M. Zaharia, et al. “Spark: Cluster Computing with Working Sets”, HotCloud’10.

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

[8] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. “Sparrow: distributed, low latency scheduling”. Proceedings SOSP '13.

[9] I. Sadooghi, K. Wang, S. Srivastava, D. Patel, D. Zhao, T. Li, and I. Raicu, “Fabriq: Leveraging distributed hash tables towards distributed publish-subscribe message queues,” IEEE/ACM International Symposium on Big Data Computing (BDC), 2015.

[10] T. Li, X. Zhou, et. Al. “ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” in Proceedings of the IEEE IPDPS, 2013.

[11] L. Thai, B. Varghese, and A. Barker, “Executing bag of distributed tasks on the cloud: Investigating the trade-offs between performance and cost,” Proceedings of CloudCom, 2014.

[12] 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.

[13] M. Schwarzkopf, A. Konwinski, M. Abd-ElMalek, and J. Wilkes. Omega: flexible EuroSys ’13, pages 351–364, New York, NY, USA, 2013. ACM.

[14] M. Frigo, et. al, “The implementation of the Cilk-5 multithreaded language,” *In Proc.* (PLDI), pages 212–223. ACM SIGPLAN, 1998.

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

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

[17] J. Dinan et. al. “Scalable work stealing,” In Proceedings of the HPDC, 2009.

[18] K.Wang, A. Rajendran, and I. Raicu. "MATRIX: Many-task computing execution fabric at exascale". 2013. http://datasys.cs.iit.edu/projects/MATRIX/index.html

[19] 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.

[20] K. Brandstatter, T. Li, X. Zhou, I. Raicu. Novoht: a lightweight dynamic persistent NoSQL key/value store. In: GCASR’ 13, Chicago, IL 2013

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

[22] I. Sadooghi, S. Palur, et al. “Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing”, Proceedings of CCGRID, 2014.

[23] 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.

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

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

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

[27] I. Sadooghi, J. Martin, T. Li, I. Raicu, et. al. “Understanding the Performance and Potential of Cloud Computing for Scientific Applications”, IEEE Transactions on Cloud Computing (TCC), 2015

[28] I. Raicu, I. Foster, and Y. Zhao, ” Many-task computing for grids and supercomputers”, In proceedings of MTAGS 2008,

[29] 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.

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

[31] H. Eslami, A. Kougkas, et. al. "Efficient disk-to-disk sorting: a case study in the decoupled execution paradigm." Proceedings of Workshop on Data-Intensive Scalable Computing Systems, 2015.