Analyzing Spark Scheduling And Comparing Evaluations On Sort And Logistic Regression With Albatross

Henrique Pizzol Grando
University Of Sao Paulo
henrique.grando@usp.br

Sami Ahmad Khan
Illinois Institute of Technology
skhan75@hawk.iit.edu

Iman Sadooghi
Illinois Institute of Technology
iman.sadooghi@gmail.com

Ioan Raicu
Illinois Institute of Technology
iraicu@cs.iit.edu

ABSTRACT

Large amounts of data that needs to be processed nowadays, have led to the Big Data paradigm and the development of distributed systems. In order to facilitate the programming effort in these systems, frameworks like Spark [10] were created. Spark abstracts the notion of parallelism from the user and ensures that tasks are computed in parallel within the system, handling resources and providing fault tolerance. The scheduler in Spark is a centralized element that distributes the tasks across the worker nodes using a push mechanism and it dynamically scales the set of cluster resources according to workload and locality constraints. However, in bigger scales or with fine-grained workloads, a centralized scheduler can schedule tasks in a rate lower than the necessary, causing response time delays and increasing the latency.

Various frameworks have been designed with a distributed scheduling approach, one of which is Albatross [7], a task level scheduling framework that uses a pull based mechanism instead of traditional push based of Spark, that uses a *Distributed Message Queue* (DMQ) for task distribution among its workers.

In this paper, we discuss the problems of centralized scheduling in Spark, show different distributed scheduling approaches that could be a fundamental idea for a new distributed scheduler on Spark and we perform empirical evaluations on Sort and Logistic Regression in Spark to compare it with Albatross.

1. INTRODUCTION

Spark is one of the Data Analytics framework proposed to particularly solve the problem of faseter data processing at larger scales by storing data in memory in order to analyze large volumes of data in seconds. It was developed aiming to facilitate the programming for distributed system. The major abstraction in Spark is the *Resiliant Distributed Dataset* (RDD), a dataset that spams many nodes and is divided in partitions, each of them possibly stored in different worker nodes. It recomputes the data automatically based on the Directed Acyclic Graph (DAG) information from the scheduling systems, thus helps in managing data-loss during the execution of the jobs.

Although Spark presents a better performance than MapReduce for iterative and CPU-bound tasks, as the number of worker nodes and cores for each machine increases, the number of tasks/second that are processed become much higher than the number of tasks that can be scheduled by the Spark centralized scheduler. This means that when sub second tasks are running, scheduling decisions must be made at a very high throughput. This causes scheduling delays with a centralized scheduler. (*Figure 1*).

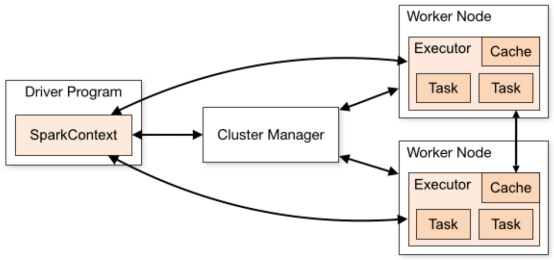

Figure 1: The centralized scheduler in Spark[11].

The usual scheduling rate for current centralized schedulers is around 2,500tasks/second. This rate can easily be surpassed by current distributed systems. "Amazon EC2 instances (m4.10xlarge) have 40 cores: a single job running on 8 workers (320 cores) can execute 2,000 tasks/second, close to the limit of a modern controller" [5].

Distributed approaches for the Spark scheduler were made [3, 6], but only at Application level, which also may become a problem when you have a fine-grained enough workload for a given application. Since in these systems each application is only scheduled by a single centralized scheduler, this scenario might lead to a scheduling rate lower than the execution rate (tasks processed by the worker nodes per second), which will cause queueing delays and a bigger latency.

2. BACKGROUND

A large variety of frameworks and engines used nowadays in distributed systems rely on previous work as building blocks. One of the systems to be explored in this paper is Albatross [7]. Albatross depends on two other projects in this area. For the sake of completeness a quick overview on these projects is provided.

2.1 ZHT

ZHT is a zero-hop distributed hash table. ZHT was developed aiming to be a building block for distributed job management systems at extreme scales. In order to attend

that demand, it's a light-weight, fault tolerant through replication, persistent (through its own non-volatile hash table: NoVoHT) and scalable system. With these properties ZHT is able to deliver high availability, good fault tolerance, high throughput, and low latencies. It's API provides four major methods: *insert*, a *remove*, *lookup* and *append* [2].

2.2 Fabriq

Fabriq is a distributed message queue built on top of ZHT. As such, it provides all the services available on ZHT including persistence, consistency, and reliability. The messages on Fabriq can be distributed across all the supercomputer nodes, allowing for parallel access to the queue. One important feature present in Fabriq is the guarantee that each message will only be delivered once [8].

3. SPARK STRUCTURE

The Cluster Manager element in the Spark architecture can be currently chosen among YARN [9], Mesos [1] and Standalone [12]. For the purposes of this study, the Standalone mode will be considered, since it simplifies the analysis with no loss of precision.

3.1 Jobs and Tasks

In order to better understand the functioning of the Spark Scheduler, some knowledge of how jobs and tasks are devised within an application is necessary.

A typical Spark application consists of a load operation from a file (from a regular file system, HDFS or similar) to an RDD where data is stored and split across different partitions. These partitions will each be handled by a different task. The task is responsible for performing the transformations, e.g. applying the user function during a Map.

Two different types of transformation can be seen in Spark. Narrow transformations (map, filter, reduce) are the ones that can be sequentially performed in a single partition. Wide transformations (sortByKey) depend on data spammed across different partitions. Therefore, wide transformations introduce a barrier on the application execution, where all partitions need to arrive before the next transformation can be performed (*Figure 2*).

These barriers are the delimiters for a stage. A stage in Spark is the equivalent of a job. This job contains a set of tasks (each of the partitions' tasks) that are scheduled in the worker nodes.

3.2 Scheduler

The main piece of the Spark scheduler is the DAG scheduler it contains the DAG that keeps track of the RDDs and their transformations. The DAG Scheduler is responsible for submitting stages as jobs to the Task Scheduler, which will then schedule the tasks across the worker nodes.

The Task Scheduler keeps track of the tasks being executed handling failures (resubmitting tasks if necessary) and mitigating stragglers. In case a whole stage is lost, the Task Scheduler asks the DAG scheduler for it to be recomputed and submitted.

Both the DAG and Task Scheduler are available in the single centralized cluster manager node. So all the scheduling decisions and operations are performed by this one node, which becomes a bottleneck in fine-grained applications and bigger scales.

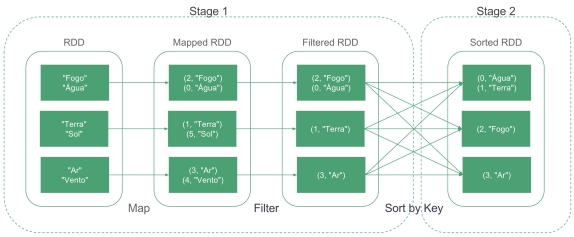

Figure 2: Narrow dependencies are kept in the same stage. Wide dependencies transformations require a barrier.

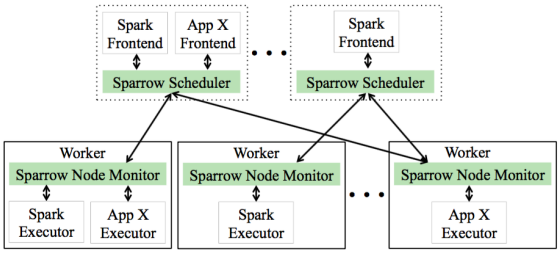

Figure 3: Sparrow architecture: each Spark application is linked to a single scheduler [3].

4. DISTRIBUTED SCHEDULING

We'll look into two different types of distributed scheduling techniques: application level (the word job is not used to avoid confusion with its meaning as a Spark stage) and task level.

4.1 Application Level

4.1.1 Sparrow

Sparrow is a distributed scheduler on application level built to work with Spark. Although, the full integration between the two systems was never made. There's only one old forked version of Spark that has an implemented plugin to accommodate Sparrow [13, 14] which doesn't support all the latest software available for big data computing.

The Sparrow architecture consists of schedulers distributed across different nodes each of them directly connected to a different Spark driver (*Figure 3*). That is, for each Spark application, only one scheduler is used. This approach sidesteps the scheduler bottleneck by giving different users, different scheduling nodes.

Although, if you have a fine-grained enough workload, even a single application can become a problem, since all of the tasks would be scheduled through the same node. Besides that, another disadvantage of this technique is that schedulers doesn't share any information, which makes the data in each application to be available only inside the application itself.

The Sparrow schedulers use batch sampling and late binding (an adaptation of the power of two choices load balancing technique [4]) in a way to improve the scheduling for a single node, which brings the performance near to the performance of an omniscient scheduler (which assumes complete infor-

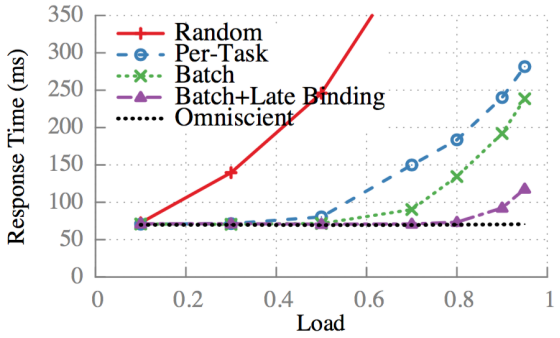

Figure 4: Different scheduling techniques compared. Batch Sampling with Late Binding boost significantly the scheduler performance [3].

mation about the load on each worker node) (*Figure 4*).

Using batch sampling the schedulers probe a number of machines proportional to the number of tasks contained in the job received (usually 2 machines/task) and instead of receiving their workload information or sending tasks it actually sends a reservation to the worker queue (late binding).

The reservation acts as a placeholder for the task, when it gets popped out of queue the worker will request a task to the corresponding scheduler. When a number of worker nodes equal to the number of tasks that need to be scheduled respond to the reservation, the scheduler will cancel the remaining reservations.

Besides the improvement on scheduling over a single node, the main problem is not addressed, but rather postponed by making the scheduling distributed across different users/ application, which still might be a problem for really big or fine-grained applications.

4.1.2 Hopper

Hopper is a decentralized speculation-aware application level scheduler built on top of Sparrow. It has the same basic characteristics that Sparrow has, namely one scheduler for each Spark application, batch sampling and late binding. On top of these strategies, Hopper implements some optimizations and modifications to ensure better scheduling decisions.

Among the optimizations made by Hopper it's worth noting that speculation tasks are treated specially and as part of the scheduling technique applied, which means these will be included in the normal scheduling decisions and not handled by some special heuristic like best-effort speculation (schedule them whenever there's an open slot) or budgeted speculation (have a fixed number of slots reserved for speculation tasks).

Best-effort speculation may lead to increased latency on task completion, since the speculation tasks depend on new slots to be vacant before they can start executing *Figure 5*. On the other hand, with budgeted speculation, one could reserve sufficient slots to execute speculation tasks as soon as they're needed, but when no speculation tasks have to be processed, those slots would be idle and CPU cycles, wasted *Figure 6*. The approach adopted by Hopper of including the speculation tasks on the scheduling decisions is better than both of the previous ones and also optimal *Figure 7*.

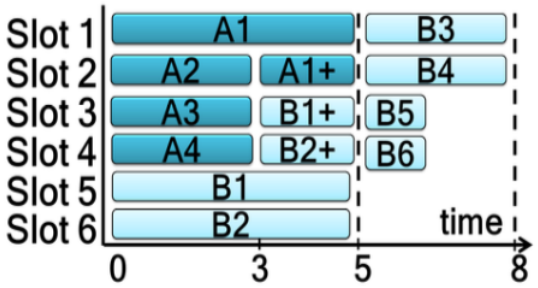

Figure 5: Best-effort approach for speculation tasks scheduling [6].

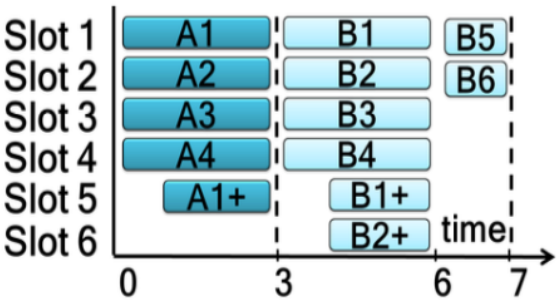

Figure 6: Budgeted approach for speculation tasks scheduling [6].

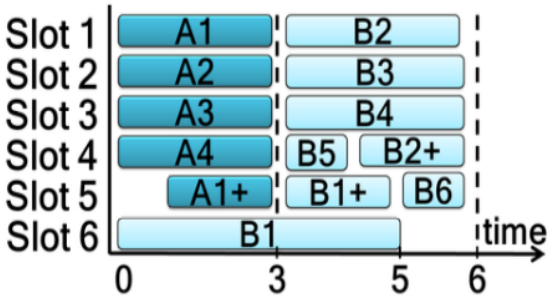

Figure 7: Speculation-aware job scheduling employed by Hopper [6].

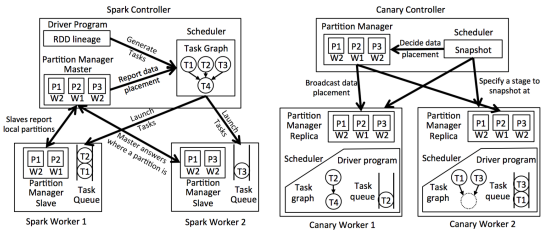

Figure 8: Comparison between Spark controller and Canary controller. Canary distributes task generation and dependency analysis to workers.[5].

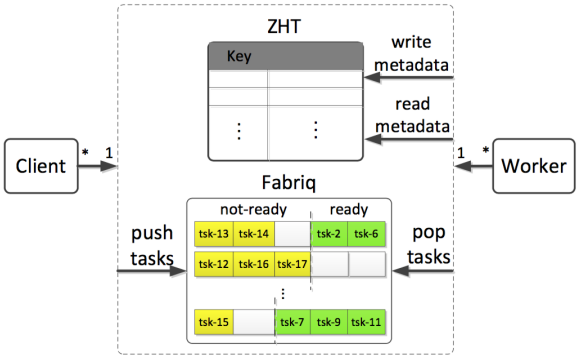

Figure 9: Two major components of Albatross: Fabriq serving as pool of tasks and ZHT for storing metadata [7].

4.2 Task Level

When considering the task level decentralized schedulers below it's worth mentioning that neither of them is directly related to Spark. Nevertheless, they implement decentralized scheduling techniques that are worth considering for eventual future implementations of decentralized schedulers on Spark.

4.2.1 Canary

Canary is a distributed scheduler architecture for cloud data processing systems. The main idea behind it is that the workers can generate tasks and execute them according to the data partitions they have in memory.

There's a central element in Canary responsible for computing and updating the partition map, specifying how partitions are distributed across the workers. The tasks that are generated and executed by the workers depends on partitions assigned to that worker, in this way, the central controller implicitly schedule tasks. In order to create and schedule tasks, each worker has a copy of the driver program *Figure 8*.

Since the driver program is distributed across all the worker nodes, there's no scalar variable (as a counter) in Canary, all variables are datasets. If this restriction were to be lifted, there would be a copy of the variable in each of the worker nodes.

Aside from computing the partition map, the central controller is also responsible for distributing the partition map to workers, coordinate worker execution so partition migration is properly performed when the partition map is changed and decide when workers should store a snapshot of their partition to non-volatile storage to provide failure recovery.

Given that, all other operations to be performed, namely spawn tasks and compute schedule dependencies, enable data exchange between workers, maintain a consistent view of execution and migrate partitions, are done by workers.

Canary assumes that the set of data changes slowly and there are few migrations of data (which is common for a CPU-bound application). If these assumptions are not respected the controller could be overloaded and latency increased.In a well-behaved job, the controller would only collect performance information. This idleness tries to avoid the bottleneck that the controller would become if more responsibilities were assigned to it.

4.2.2 Albatross

Albatross is a task level distributed scheduling and execution framework. In order to achieve scalability the Albatross system lacks a central controller, differently than Canary the scheduling effort is performed by the workers and there's no central element of coordination.

In order to achieve that, Albatross uses ZHT and Fabriq that interact with Client and Worker drivers. Fabriq is used as a pool of tasks to where Clients push tasks and Workers pull tasks when needed. ZHT is used to store metadata information that is read and written by Workers, this metadata could be, e.g., the data location and the workload DAG *Figure 9*.

Since all elements in the system are fully distributed, a node in Albatross could have any of the components or even a combination of them. Indeed each worker driver will have access to a local instance of both a ZHT and a Fabriq server.

To start the process, the user provide the job information and the dataset to the Client driver (or drivers). The Client splits the data across the Worker driver nodes and based on the information received generates the tasks and the dependency DAG. It then submits the tasks to Fabriq using a uniform hashing function to ensure load balance.

In each worker there are two different queues, the main one is used to pull tasks out of Fabriq and the other one is the local queue, to which the tasks are transferred if the data needed for that task is local to the node. Regarding the remaining tasks in the main queue, they're preferably transferred to the node where the data is located, but if that node is already overloaded, the data is fetch and the task is executed locally.

This kind of approach avoids the bottleneck created by a centralized element, once everything is distributed. On the other hand, depending on the nature of the application, the Workers may need to exchange information or, in the case of a highly loaded cluster, data too frequently which could lead to network delays.

The last aspect of implementation that's worth being mentioned is the way the dependencies are handled through Fabriq. Each task has two fields that pCount and ChildrenList, that is, the number of tasks that still need to be executed before the task in question; and a list with all the tasks that depend on it. Workers only pull tasks with pCount=0 and when a task is executed, its list of Children is traversed and their pCount decremented.

5. IMPLEMENTATION

We implemented Sort and Logistic regression, with same exact details, on Spark and Albatross to evaluate its performance on varying nodes cluster with varying workloads. First, we briefly describe the two metrics that we have used for the comparison. Then, we measure the efficiency, throughput and latency while varying the granularity of the workloads.

5.1 Sort

The Sort benchmark measures the amount of time taken to sort a specific size of randomly distributed data over a distributed cluster. In our case we have used varying input sizes according to number of nodes in both Spark and Albatross cluster and compared the time taken on both the frameworks. The input for the Sort is generated using the Gensort[15] application and consists of skewed key-value pairs i.e. 10 bytes of key and 90 bytes of value in a row. The sorting is done on the basis of Key.

5.2 Logistic Regression

Logistic Regression[16] is an algorithm in Machine Learning for Classification. Classification involves looking at data and assigning a class (or a label) to it. Usually there are more than one classes, but in our case, we have tackled binary classification, in which there at two classes: 0 or 1.

Essentially what we do, is draw a 'line' through our data, and if a data point (or sample) falls on one side, assign it a label 0, or if it falls on the other side of the line, give it a label 1. A *Labeled point* is a local vector associated with a label/response. In MLlib, labeled points are used in supervised learning algorithms and they are stored as doubles. For binary classification, a label should be either 0 (negative) or 1 (positive).Classifying all the points requires a certain number of iterations for more accurate classification of the data points. In our case we have calculated the total time to classify the data points with 50 iterations.

An explanation of logistic regression is done by means of standard logistic function or the sigmoid function. The sigmoid function can take input from any negative to positive infinity whereas the output always takes values between 0 and 1 and hence can be translated as probability. The sigmoid function is defined as,

$$g(z) = \frac{1}{1 + e^{-}z}$$

Here is a plot showing $g(z)$,

Machine Learning 'fits' this line using a optimization algorithm (usually Gradient Descent or some form of it), such that the error of prediction is lowered. In our case we have used *Stochastic Gradient Descent* (SGD) model for our classification. SGD is an iterative optimization algorithm that can be applied for discriminative learning of linear classifiers under convex loss functions such as Support Vector Machines and Logistic Regression. These types of functions often arise when the full objective function is a linear combination of objective functions at each data point.

6. EXPERIMENTAL EVALUATION

We evaluate and compare Spark and Albatross by running Sort and Logistic Regression on both with exact details. We ran these metrics on varying cluster nodes with varying workloads. Following table shows the scaling of workload with number of nodes for both Sort and Logistic Regression:

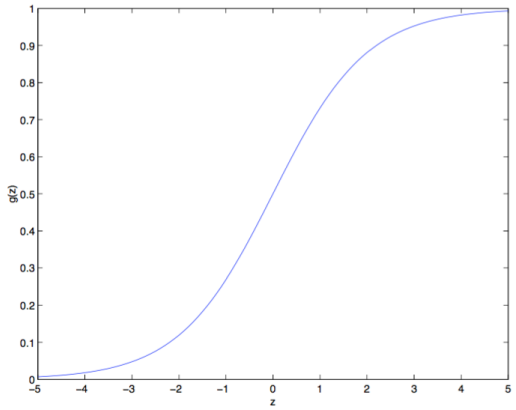

Figure 10: Notice that $g(z)$ tends towards 1 as z $\rightarrow \infty$, and $g(z)$ tends towards 0 as z $\rightarrow -\infty$.

Nodes	Workload Size (in GB)
1	5
2	10
4	20
8	40
16	80
32	160
64	320

Table 1: Table to test captions and labels

We measure latency and throughput while running benchmarks on both Spark and Albatross

6.1 Testbed and Configurations

The experiments were done on m3.large instances for Sort, which have 7.5 GB of memory, 32 GB local SSD storage and High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors with 2 vCores and m3.xlarge instances for Logistic Regression, which have 15 GB of memory, 80 GB local SSD storage and Intel Xeon E5-2670 v2 (Ivy Bridge) Processors with 4 vCores. The workload was scaled as 5 GB per node as shown above in *Table 1*

6.2 Latency

In order to compare the scheduling cost of Spark with Albatross, we calculated the total time taken for the Sort and Logistic Regression benchmarks to run on Spark and then compare them with Albatross.

6.2.1 Sort

For sort, we first compared our values of Sort on Spark with the existing values of another student, Geet Kumar, from Illinois Institute of Technology and found out that our code was more optimised than the later and ran quite faster than his. The following comparison is shown in *Figure 11* and *Figure 12*

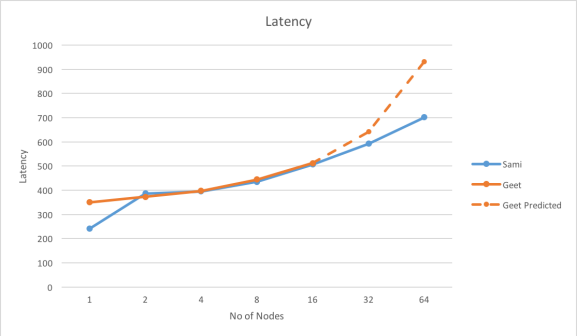

Figure 12: Line comparison of average latency for in-memory Sort tasks

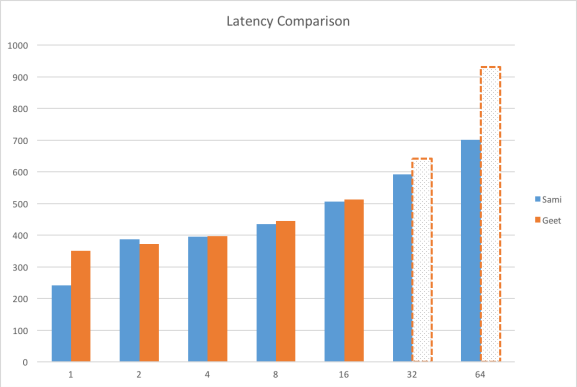

Figure 11: Side by side comparison of average latency for in-memory Sort tasks

From above figures, it is evident that our version of Sort, represented in blue bar runs comparatively better for 1,4,8 and 16 nodes and the performance improves noticeably for 32 and 64 nodes. Hence it can be predicted that our version of Sort may improve latency by a lot with increase in number of nodes.

6.2.2 *Logistic Regression*

For Logistic Regression, we have calculated the total time (in msec) to classify (train) the data points with 50 iterations, using the Stochastic Gradient Descent model in Spark and Albatross. We have scaled our experiments from 1 to 64 nodes and 5Gb to 320Gb workload sizes respectively. The latency is calculated for time taken to train per GB of data for 1 to 64 nodes.

Figure 13 shows the average latency of running iterative tasks on Spark, scaling from 1 to 64 nodes. Ideally, on a system that scales perfectly, the average latency should stay the same. And it can be seen that logistic regression in Spark runs very close to the ideal case i.e. we observe a curve that approximately looks like a straight line.

6.3 Throughput

In order to analyse the task submission and scheduling performance of the frameworks, we measured the total timespan for running Sort and Logistic Regression on varying nodes with varying workload size and then divide the total workload size (in Mb) with the total time to find the

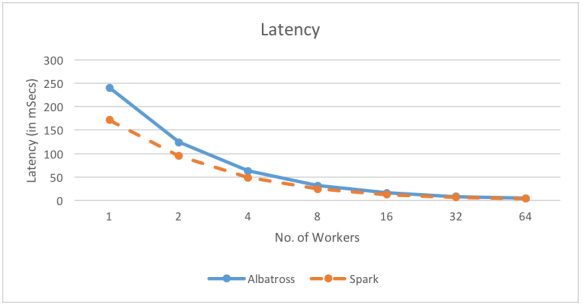

Figure 13: Average Latency for Logistic Regression on varying nodes for 50 iterations

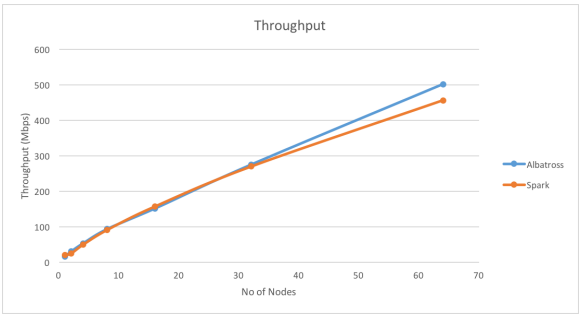

Figure 14: Throughput comparison for Sort on Spark and Albatross

throughput for that specific node. The throughput is defined as the number of tasks processed per second (tasks per second),

$$Throughput(T) = \frac{WorkloadSize(inMb)}{TotalTimetaken(inseconds)}$$

6.3.1 *Sort*

The throughput performance for sort on Spark and Albatross is shown in *Figure 14*

In *Figure 14*, we can see that Albatross tends to perform slightly better on larger scales (linearly) i.e. Spark is almost an order of magnitude slower than Albatross for 32 and larger nodes. Spark's centralized scheduler starts to saturate after 64 nodes with a throughput of 456.49 Mb per second. Spark's throughput performance scales 1.25 times to previous value, when number of nodes are increased. Whereas it increases by almost 1.70 times to previous value in Albatross, which can almost be approximated as double i.e. throughput increases as 2x with increasing number of nodes. Hence it can be predicted that Albatross performs much faster on larger nodes, scaling workloads effectively and efficiently.

From *Figure 15*, we can see the difference in performance of the three frameworks, namely Spark, Hadoop and Albatross for sort on different partition sizes on 64 nodes. The competition between Spark and Albatross is very close and Hadoop is nowhere close to them. Albatross is at an average 3.13% faster than Spark overall and 5.89% faster for large scales i.e. 32 and 64 nodes, and 56.63% faster than Hadoop.

6.3.2 *Logistic Regression*

The throughput performance comparison for Logistic Regression on Spark and Albatross is shown in *Figure 16*

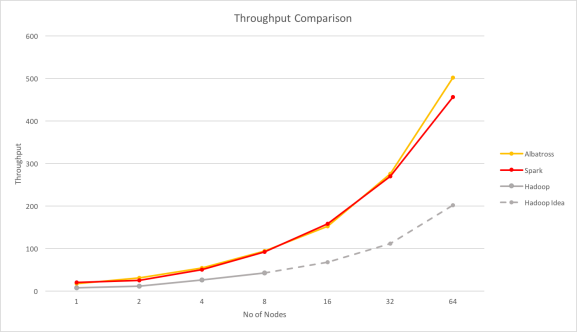

Figure 15: Throughput comparison for Spark, Albatross and Hadoop for Sort

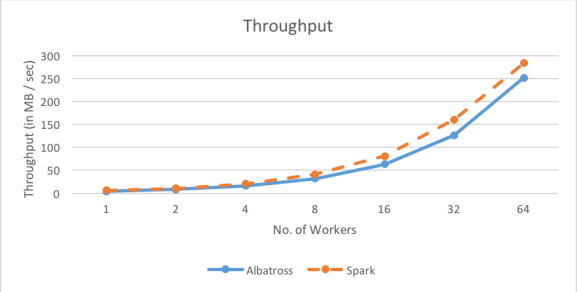

Figure 16: Throughput for Logistic Regression on varying nodes in Spark

Figure 16 shows that Logistic Regression on Spark performs slight better than Albatross. This is due to the highly optimized RDDs and Scikit-Learn libraries for Logitic Regression in Python.

7. CONCLUSIONS

The Spark centralized scheduler is a bottleneck for very large distributed systems. Existing distributed implementations like Sparrow, which is compatible with Spark try to solve this problem on application level, which is still not an optimal solution, since a fine-grained enough workload could surpass the single scheduler capacity. Therefore, a distributed scheduler on task level, inspired in systems like Albatross, for Spark is needed in order to provide a high rate of scheduled tasks/second that can keep up with demanding workloads.

Albatross seems to perform quite better than Spark, as seen in *Figure 16* for the Sort by an order of magnitude. But Logistic Regression on Spark came out to be slightly better than Albatross because of highly optimized MLib libraries available for Machine Learning on Spark. And since Logistic Regression for Albatross was written from scratch in C, the mappers and reducers were not as optimized to RDDs which caused the delay in convergence.

Since Albatross still gives high competition to Spark for most of the evaluation. It is quite possible that with more optimized code and configurations, Albatross would perform much better than Spark. Hence, by means of this paper I can state that the Pull based mechanism in Albatross is a better option for distributed scheduling on task level than a Push based Spark.

8. REFERENCES

[1] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22.

[2] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 775–787.

[3] 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*, pages 69–84. ACM.

[4] G. Park. A generalization of multiple choice balls-into-bins. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 297–298. ACM.

[5] H. Qu, O. Mashayekhi, D. Terei, and P. Levis. Canary: A scheduling architecture for high performance cloud computing.

[6] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 379–392. ACM.

[7] I. Sadooghi, G. Kumar, K. Wang, D. Zhao, T. Li, and I. Raicu. Albatross: an efficient cloud-enabled task scheduling and execution framework using distributed message queues. In *Tech report*.

[8] I. Sadooghi, K. Wang, D. Patel, D. Zhao, T. Li, S. Srivastava, and I. Raicu. FaBRiQ: Leveraging distributed hash tables towards distributed publish-subscribe message queues. In *2015 IEEE/ACM 2nd International Symposium on Big Data Computing (BDC)*, pages 11–20.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16. ACM.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. 10:10–10.

[11] Cluster mode overview - spark 1.6.2 documentation.

[12] Job scheduling - spark 1.6.2 documentation.

[13] kayousterhout/spark/tree/sparrow - github repository.

[14] Sparrow users - google groups.

[15] Gensort - random data generator for sort bechmarks.

[16] Logistic regression.