Optimizing Task Management Using Distributed Load Balancing and Work Stealing with Matrix over YARN

Anilraj Chennuru
Dept. of Computer Science
Illinois Institute of technology
Chicago IL, USA
achennur@hawk.iit.edu

Meenakshi Mulinti
Dept. of Computer Science
Illinois Institute of technology
Chicago IL, USA
mmulinti@hawk.iit.edu

Petter Castro
Dept. of Computer Science
Illinois Institute of technology
Chicago IL, USA
pcastro@hawk.iit.edu

Praneender Vuppala
Dept. of Computer Science
Illinois Institute of technology
Chicago IL, USA
pvuppal2@hawk.iit.edu

***Abstract*—For several years, researchers have invested great efforts to improve Hadoop framework through implementations of different architectures to be able to handle large scale data processing. YARN is the latest architecture on Hadoop, keeping decoupled components to maintain scalability. However, data is growing so fast at extreme scales and computing systems need to adapt to it and YARN may not be suitable because of a centralized design in the scheduling system. In this paper, we present MATRIX as a distributed scheduler which utilizes work stealing to achieve load balancing. We override the default centralized scheduling system on the current resource manager of YARN. By doing this, we look forward to get a higher throughput, efficiency and speedup of the jobs processing at extreme scales in Hadoop.**

I. INTRODUCTION

Today?s science produces data at a rapid and unpredicted rate. Handling large amounts of data while achieving higher throughput and low latency is a challenge. Through the Hadoop implementation and framework of YARN have allowed MapReduce to scale to higher extent. As in near future predictions are that we will reach exascale computing, but the centralized architecture of YARN prevents Hadoop achieving higher throughput and low latency at this extreme scale of computing. Implementing the MATRIX scheduling system on top of YARN makes Hadoop fully distributed and also achieve higher throughput, low latency. This paper provides evaluation comparing the results of running applications on Matrix and YARN.

A. YARN

YARN is the Hadoop's compute platform which decouples the programming model from the resource management by splitting responsibilities among its components [7]. First, it has one Resource Manager for the entire cluster with two components: a scheduler to handle all the tasks, and an application manager to control and monitor an application master. Then, it also contain a per-application Application Master which is the one who talk with the Resource Manager for negotiating resources and submit the tasks to every data node. Finally, a Node Manager per slave which is the 'worker' daemon in YARN and it monitors all the tasks execution in each data node. Figure 1 illustrate these components in the YARN architecture.

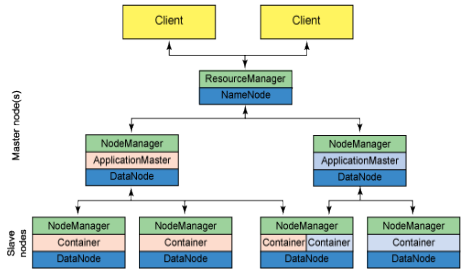

Fig. 1. Architecture of YARN

So, by submitting a job, all of these components take action in this way:

- Client submit a job to the Resource Manager.
- Resource Manager scheduler search and allocate one Application Master in a container with enough resources.
- Application Master get metadata information from HDFS.
- With this data, Application Master negotiates with the Resource Manager scheduler to know which data nodes will execute the tasks.
- Application Master submit the tasks to the data nodes given from the Resource Manager.

B. MATRIX

Matrix is a distributed task execution framework that utilizes work stealing technique. The architecture comprises of a scheduler, an executor and a ZHT server. The MATRIX architecture and the components, along with the communication signals among the components are shown in Figure 2. Each compute node runs a scheduler, an executor and a ZHT server [1]. The executor is a separate thread in the scheduler. All the schedulers are fully connected with each one knowing all of others. The client is a benchmarking 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 waiting tasks, it initializes work stealing to steal tasks from neighbors. The tasks being stolen would be migrated from the victim to the thief. In this scenario, task migration doesn?t involve moving data. We will show how work stealing contributes to data-aware scheduling later [1].

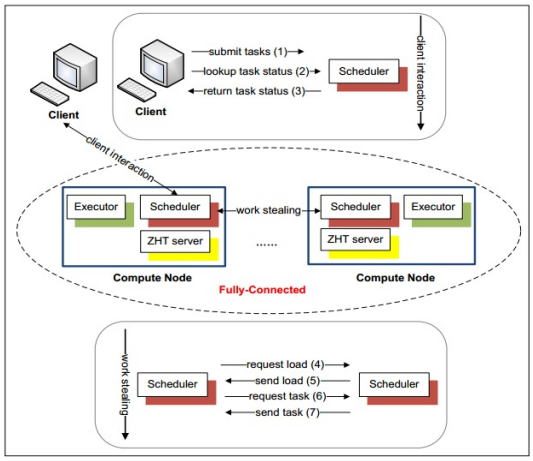

Fig. 2. MATRIX components and communications among them

The executor keeps executing tasks and when there are no more tasks to execute on a particular scheduler, it ?Steals? the tasks from the neighboring schedulers. All the schedulers will be aware of all others. We use ZHT to keep the task-metadata in a distributed, scalable and fault-tolerant way. The tasks are submitted in the form of batches to improve the per client throughput. Performance of MATRIX is compared to sparrow and CloudKon. It showed a high throughput and efficiency and has a near perfect load balancing while submitting tasks. All the tasks will be put in the wait queue initially. The task dependency is stored in ZHT in the following way. Each task has two fields: a counter, and a list of child tasks. ”key” is the task-id, the two fields are part of the value. The counter represents the number of parent tasks that should complete before this particular task can be executed. The list of child tasks is the list of tasks that are waiting for this particular task to complete. A program P1 would keep checking every task in the wait queue to see whether the dependency conditions for that task are satisfied or not by querying ZHT server [10]. The dependency conditions would be satisfied only if all the parent tasks have been finished (the counter is 0). Once a task is ready to run, it would be moved from the wait queue to the ready queue by P1. After finishing the execution of a task, the task is then moved to the complete queue. As long as the ready queue is empty, the scheduler would do work stealing, and try to steal tasks from randomly selected neighbors. Only the tasks in the ready queue can be stolen. Tasks in the wait queue would be marked as unreachable for other schedulers. For each task in the complete queue, another program P2 in the executor is responsible for sending a completion notification message to ZHT server for each child task. The ZHT server would then update the dependency status of each child of that particular task by decreasing the counter by 1. As long as the dependency counter of a task is 0, the task would be ready to run. Whenever the status of a task is changed, the scheduler would update the changing information by inserting the updated information to ZHT server for that specific task [10].

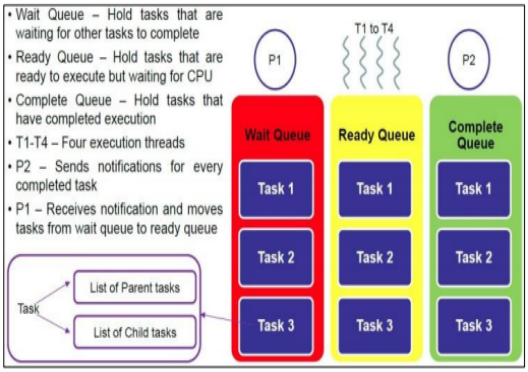

Fig. 3. Execution Unit in MATRIX

II. Problem Statement

Predictions are that we will reach exascale computing with hundreds of thousands to millions of nodes and up to a billion threads of execution in less than a decade [8], therefore achieving low latency, high throughput and efficiency is a concern with present centralized scheduling system. To handle problems with Big data, centralized architecture is not a best option. Therefore present YARN architecture is not suitable for exascale systems. So we propose a new scheduling system MATRIX on top of YARN making it fully distributed which produces low latency and high throughput.

III. Proposed Solution

YARN architecture has 3 main components, a global Resource Manager, a Node Manager per slave and a Application Master per application [7]. These are crucial in the architecture of YARN, but the components interacting with the scheduling system are in which we focus on our solution. First our focus is on the Application Master as it is responsible for negotiating resources with the Resource Manager to execute the jobs, therefore we remove communication between them for negotiating resources because it is not required anymore, as resource manager sends the tasks to Application master where we put MATRIX Client in it. Matrix client in application master submit tasks to matrix schedulers randomly present on each node manager. We define MATRIX client in Application Master so it can submit the tasks to any compute node where matrix scheduler is placed on Node managers. To do this, we implement the communication interfaces between MATRIX schedulers and the Application Master. We add MATRIX as a distributed scheduler, placing the three components of it (Scheduler, Executor and KVS) on each of the Node manager, having now a distributed scheduling system in our architecture.

Finally, as the communication with the HFDS and application master is removed, we need to add this communication in MATRIX, so we implement the interface to communicate each scheduler with HDFS, and that way MATRIX enables data-aware scheduling.

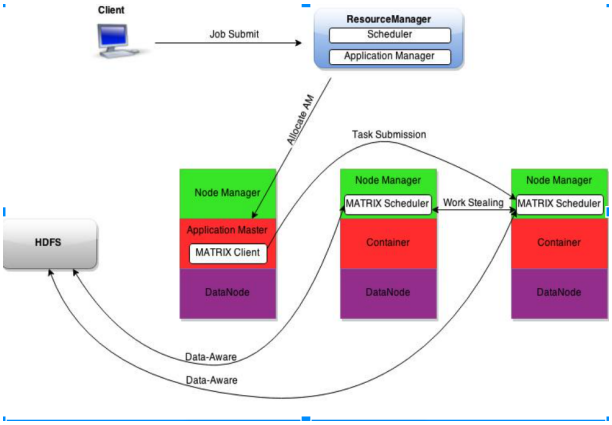

Fig. 4. Proposed Framework

IV. Related Work

In order to address issues like latency and throughput for big data, companies like Facebook came up with their own scheduling system on top of Hadoop File System(HDFS) known as Corona. Corona tries to overcome high latency and low throughput. Though it may not be a better option for exascale systems as facebook's corona is a centralized architecture which may not be suitable for exascale computing. Hadoop?s JobTracker is responsible for both cluster management and job-scheduling, but has a hard time keeping up with both tasks as clusters grow and the number of jobs sent to them increase. To solve these issues apache implemented new scheduling scheme MapReduce v2 also know as YARN which tries to address same problems as Corona. Since it has a centralized architecture it may not be suitable for exascale systems. There is another system Mesos which is a more of a research project by Apache, unlike YARN which was created as a necessity to move the Hadoop Mapreduce to the next iteration and life cycle. The primary difference between Mesos and Yarn is going to be its scheduler. In Mesos, when a job comes in, a job request comes into the Mesos master, and what Mesos does is it determines what resources are available, and it makes offers back. Those offers can be accepted or rejected whereas in YARN, it?s Monolithic. So, basically mesos is a global resource manager for the entire data center [9]. Therefore we propose a new scheduling system MATRIX on top of HDFS which is a distributed task execution fabric (i.e. MATRIX) that utilizes work stealing to achieve distributed load balancing, and a distributed key-value store to manage task metadata. Matrix outperforms sparrow (a centralized architecture) and Cloudkon (Distributed Scheduling System) as mentioned in [1].

V. Implementation

While integrating Matrix with Hadoop, lot of issues raised due to dependencies between HDFS and YARN. We would like to implement our architecture as a future work and we compare Matrix and Hadoop and evaluate their performance in terms of throughput and time. We present our results in evaluation section and show how MATRIX outperforms HADOOP. In this section we present how we configured Hadoop cluster and MATRIX and applications that we executed on both the clusters.

A. Setting up HADOOP:

Since it is difficult to configure manually we have written a script that installs Hadoop on all the nodes. It installs JRE and JDK and sets up path. It also configures Hadoop by taking inputs from user. The below mentioned are few important configurations that are required to set up Hadoop on multi-node cluster.

B. Configuration files:

We overwrote the default properties of the Hadoop to get the best performance out of it and compare it with matrix and evaluate it. Below lists the properties that we configured in each XML file.

- core-site.xml:This file contains the file system address of HDFS with the property name described as "fs.default.name"
- Hdfs-site.xml:Here we overwrite the default path of namenode and datanode where namenode stores FSimage and edits log and Datanode stores the data. We use property dfs.namenode.name.dir to give the path for name node to store metadata. For data node we use dfs. data node.data.dir to give the path in the data node to store the data.
- Yarn-site.xml:In this xml file we set yarn.nodemanager.aux-services as map redduce.shuffle and "yarn.nodemanager.auxservices .mapreduce. shuffle. class" as org.apache .hadoop .mapreduce.ShuffleHandle
- Mapred-site.xml: In this configuration file we set the mapreduce framework as yarn using the property "mapreduce.framework.name".

C. Setting up MATRIX

Similarly we used a script that configures Matrix upto 32 nodes as it is very difficult to configure 32 nodes manually.

D. Matrix Configuration:

Installs google c++ and c protocol buffers and sets the path. Downloads matrix source. Compile the source code of ZHT and matrix. Add all the ip addresses of the cluster to config file. As mentioned earlier we used a script to set up Matrix on multiple nodes. We give 8 inputs to the script that automates few properties at runtime.

- NumNode: Total number of nodes in the cluster. numTaskPerClient: Average number of tasks per clients

- matrixPath: This argument is used to record src path of matrix.
- zhtSrcPath: It has the ZHT src path.
- DagType: Provides the DAG type whether it is a pipeline, BOT,Fanin,Fanout.
- DagArgument: Parameter of the workload DAG. numMapTask: Number of maps for the application.
- numReduceTask: Number of reducers for the application.

We executed 4 applications Wordcount, Teragen, Terasort and GREP on hadoop up to 64 nodes and found out few drawbacks of centralized scheduling.

VI. Evaluation

In this section we will show the results that are evaluated comparing Matrix and YARN. We evaluated all results on Amazon EC2 c3-large instances.

a) : The below graph shows the time required to run 4 applications GREP, Teragen, Terasort, Word count applications on 1,4,16 and 64 nodes on Hadoop. From the figure 5, we can clearly make out that as the number of nodes increases as shown in the figure 5, the time required to compute the applications decreases by an order of magnitude for almost all the applications.

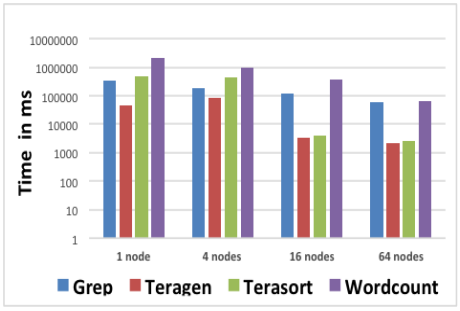

Fig. 5. Time comparison between 4 applications on Hadoop.

The below graph shows that there is not much difference in time for running teragen and terasort applications on 16 and 64 node which is one of the drawbacks of centralized scheduling which reaches its threshold much faster than distributed scheduler.

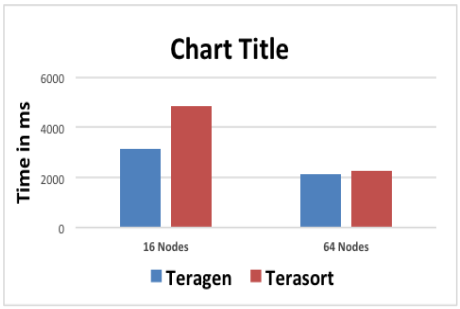

Fig. 6. Performance of Terasort and Teragen application on 16 and 64 nodes

We executed terasort and teragen on two data sets 5 GB and 10 GB and we found out that the real time difference on 16 and 64 nodes for two applications is not much. We can clearly say that for few applications changing the size of data set may not reduce the time in centralized schedulers that?s because it may not scale for applications of smaller datasets. While MATRIX improves the throughput as we scale the number of nodes and tasks as shown in fig 9 and fig 10.

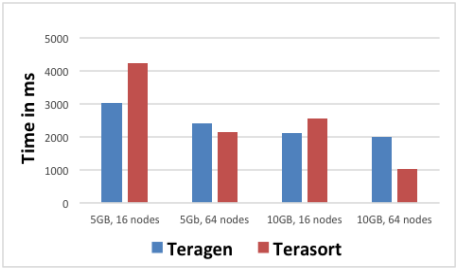

Fig. 7. Time comparison between two different datasets.

We run experiments on matrix up to 32 nodes and found out that matrix shows significant improvement in performance as we scale to 32 nodes.

b) : Figure 8 shows time taken to execute the tasks on 1, 2 and 4 nodes. We can see from the graph there is a significant decrease in time from 1 node to 4 nodes. It clearly says that matrix is highly scalable (i.e., as the number of nodes increases time to execute tasks also decreases).

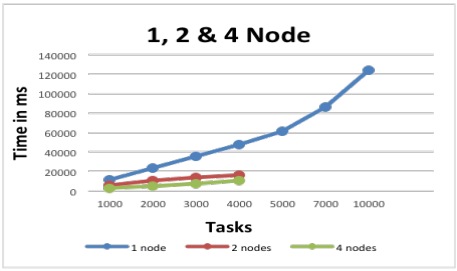

Fig. 8. Time for different number of nodes as tasks increases.

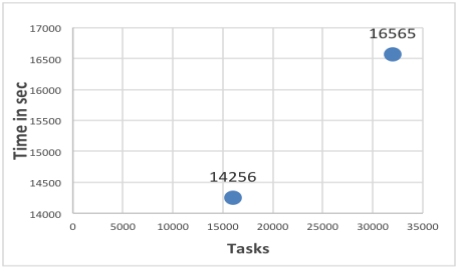

Fig. 9. Comparison for 16 and 32 nodes.

Time taken to run 16000 tasks on 16 nodes and 32000 tasks on 32 nodes are almost same from this we can say that throughput doubles when the number of nodes doubles. We

can draw that throughput is proportional to number of nodes as the number of nodes increases throughput also increases proportionally. Throughput for 32 nodes almost doubles compared to 16 nodes that is the reason why time is same for 16k tasks on 16 nodes and 32k tasks on 32 nodes as shown in the figure 11.

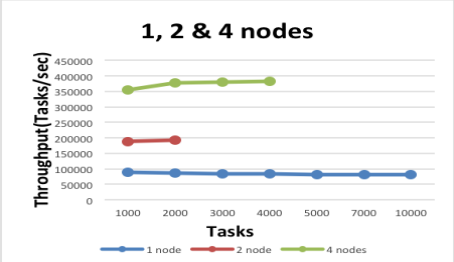

Fig. 10. Throughput for varying number of nodes

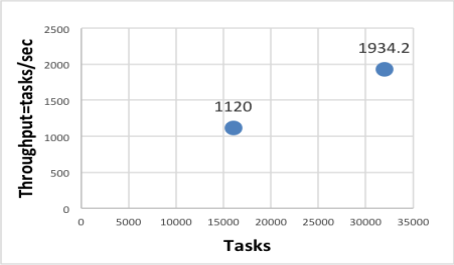

Fig. 11. Illustrates improvement in throughput as number of nodes increases

VII. Conclusion and Future Work

Our main aim is to integrate Distributed scheduling framework with HDFS to achieve high throughput and low latency. As this paper discuss and present results about how distributed scheduling achieves high throughput and low latency by running experiments on amazon EC2 instances. Our future work is to replace YARN with MATRIX in Hadoop. Extreme-scale supercomputers require next-generation Task Scheduling Frameworks to be fully distributed that can be much more scalable to deliver jobs with much higher throughput. At the same time it is necessary to have a better file system which is responsible for metadata management. So we believe HDFS possess all those features to be integrated with MATRIX, A distributed scheduling framework which uses adaptive work stealing technique. Previous results shows that MATRIX performance is more preferable than Centralized architecture in achieving high throughput and low latency.

VIII. References

- [1]. Kewang, Anupam Radrendan, Xiaobing Zhou, Kiran Ramamurthy, Iman Sadooghi, Michael Lang, Ioan Raicu, ”Distributed Load-Balancing with Adaptive Work Stealing for Many-Task Computing on Billion-Core Systems”.
- [2]. Iman Sadooghi, Sandeep Palur, Ajay Anthony, Isha Kapur, Karthik Belagodu, Pankaj Purandare, Kiran Ramamurty, Ke Wang, Ioan Raicu, ”Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High-Performance Computing”.
- [3]. MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat, Google, Inc
- [4]. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, Bigtable: ”A Distributed Storage System for Structured Data”.
- [5] .Tak-Lon (Stephen): ”MapReduce and Data Intensive Applications”.
- [6]. V. Sarkar, S. Amarasinghe, et al. ”ExaScale Software Study: Software Challenges in Extreme Scale Systems”, ExaScale Computing Study, DARPA IPTO, 2009.
- [7] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O?Malley, Sanjay Radia, Benjamin, Reed, Eric Baldeschwieler, ”Apache Hadoop YARN: Yet Another Resource Negotiator”.
- [8] Ranger C, Raghuraman R, Penmetsa A, Bradski G- ”Evaluating MapReduce for Multicore and Multiprocessing Systems”.
- [9] Ke Wang, Ioan Raicu, ”Scheduling Data-Intensive Many-Task Computing Applications in the Cloud”.
- [10]. Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang and Ioan Raicu. 2013.” ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table”, 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS’13).