A rack-aware scalable resource management system for Hadoop YARN

Timothy Moses*

Department of Computer Science,
Federal University of Lafia,
Nasarawa State, Nigeria
Email: moses@fuwukari.edu.ng
*Corresponding author

Hyacinth C. Inyiama

Department of Electronic and Computer Engineering,
Nnamdi Azikiwe University,
Awka, Anambra State, Nigeria
Email: drinyiamahc@yahoo.com

Sylvanus O. Anigbogu

Department of Computer Science,
Nnamdi Azikiwe University,
Awka, Anambra State, Nigeria
Email: dranigbogu@yahoo.com

Abstract: Big data have brought in an era of data exploration and utilisation with MapReduce computational paradigm as its major enabler. Though great efforts through the implementation of Hadoop have made computation scale to tens of thousands of commodity cluster processors, the centralised architecture of resource manager has adversely affected response time in large data centres. The developed model decouples the responsibilities of resource manager by providing another layer where each daemon called rack unit resource manager (RU_RM) carries out the responsibility of allocating resources to compute nodes within its local rack to ensure low latency for large files. The application was developed and tested with Hadoop workload benchmarks used for analysis. Two performance evaluation metrics (efficiency and average task-delay ratio) were used for comparison. Efficiency quantifies average cluster utilisation while average task-delay ratio measures average delay time. Results obtained showed that as file size increases, the developed model outperforms the existing framework.

Keywords: MapReduce; Hadoop; framework; scalable; rack-aware; resource manager; big data; rack unit resource manager.

Reference to this paper should be made as follows: Moses, T., Inyiama, H.C. and Anigbogu, S.O. (2020) 'A rack-aware scalable resource management system for Hadoop YARN', *Int. J. High Performance Computing and Networking*, Vol. 16, No. 1, pp.1–13.

Biographical notes: Timothy Moses is currently a Lecturer at the Department of Computer Science, Federal University of Lafia (FULAFIA). Prior to joining FULAFIA, he worked as Senior System Analyst/Programmer, then Lecturer at the Department of Computer Science, Federal University Wukari, Taraba State. His area of interest is distributed database management and parallel computation. He has also picked keen interest in the area of educational technology. He is a member of the Nigeria Computer Society.

Hyacinth C. Inyiama is a Professor of Computer and Control Engineering, Department of Electronics and Computer Engineering, Nnamdi Azikiwe University Awka. His research interest is in artificial intelligence and industrial automation. He has provided solutions to many industry automation problems. He is currently on sabbatical at Baze University, Abuja – Nigeria. He is a member of the Nigeria Computer Society and Computer Professionals (Registration Council of) Nigeria.

Sylvanus O. Anigbogu is a Professor of Computer Science, Department of Computer Science, Nnamdi Azikiwe University Awka. He is currently the Dean, Faculty of Physical Sciences. His research interest is in the areas of artificial intelligence, database design and management, cyber security. He has published his work in several local and international journals. He is also a member of the Nigeria Computer Society and Computer Professionals (Registration Council of) Nigeria.

1 Introduction

The growing popularity of cloud computing and advances in information and communication technology (ICT) have led to a continuous increase in the volume of data and its computational capacity has generated an overwhelming flow of data now referred to as big data (Sergio, 2015). These ever-increasing data pools have a profound impact not only on hardware storage requirements and user applications but also on the file system design, implementation and the actual I/O performance and scalability behaviour of today's IT environment (Wu et al., 2006). To improve I/O performance and scalability therefore, the obvious answer is to provide a means such that users can read/write from/to multiple disks (Dominique, 2015). Today's huge and complex semi-structured or unstructured data such as graph analytics, which have gained rapid application in e-commerce, social networks and recommendation systems (Fu et al., 2019) and smart health monitoring system, which has improved quality of healthcare services (Aboudi and Benhlima, 2017) are difficult to manage using traditional technologies like relational database management system (RDBMS) hence, the introduction of Hadoop distributed file system (HDFS) and MapReduce framework in Hadoop. Hadoop is a distributed data storage/data processing framework (Vinod et al., 2013). Hadoop was designed to process efficiently, large data volumes by linking many commodity systems so that they can work as a parallel entity (Vinod et al., 2013). The framework was designed basically to provide reliable, shared storage and analysis infrastructure to the user community. Hadoop has two components – HDFS (Konstantin et al., 2010) and the MapReduce framework (Dean and Ghemawat, 2004). The storage portion of the framework is provided by HDFS while the analysis functionality is provided by MapReduce (Konstantin et al., 2010; Dean and Ghemawat, 2004). Other components also constitute Hadoop solution suite.

The first generation Hadoop called Hadoop_v1 was an open-source of MapReduce (Bialecki et al., 2005). It has a centralised component called JobTracker that plays the role of both resource management and task scheduling. With Hadoop_v1, scalability beyond 4,000 nodes was not possible looking at the centralised responsibility of JobTracker/TaskTracker architecture (Vinod et al., 2013). To overcome this bottleneck and to promote this programming framework so that it carries other standard programming models and not just implementation of MapReduce, the Apache Hadoop community developed the next generation Hadoop called yet another resource negotiator (YARN) (Vinod et al., 2013). This newer version of Hadoop called YARN decouples resource management infrastructure from JobTracker in Hadoop_v1. Hadoop YARN introduced a centralised resource manager (RM) that monitors and allocates resources (Vinod et al., 2013).

RM exposes two public interfaces and one internal interface (Vinod et al., 2013). The public interfaces are client submitting applications and application master (AM) dynamically negotiating access to resources. The internal interface is towards the node manager's (NMs) ability for cluster monitoring and resource access management (Vinod et al., 2013). For this work, the focus is on public interfaces as it best explains an important frontier between YARN platform and various applications/frameworks running on it. RM is a global model of cluster state against the digest of resource requirements reported by running applications. AMs codify their need for resources by making one or more resource-requests each of which track the number of containers (e.g., 200 containers), resource per container (4 GB, 2 CPU), locality preference and priority of requests within the application (Apache Hadoop, 2018). Resource-requests are designed in a way that captures the full detail of users' needs and/or a roll-up version of it. RM responds to AM request by generating containers together with tokens that grant access to resources (Apache Hadoop, 2018). Once an application completes its execution, RM forwards an exit status of finished containers as reported by NMs to the corresponding AMs (Apache Hadoop, 2018). Looking at the responsibilities of RM, it is important to point out that RM is not responsible for coordinating application execution or task fault tolerance. It does not provide status or metrics for running applications (now part of AM) and it does not serve framework-specific reports of completed jobs (now delegated to a per-framework daemon). RM only handles live resource scheduling of applications with the heartbeat communication from AMs and NMs in the cluster. However, for a greater number of commodity servers and applications demand, response time from the global model of RM will be slow. It is, therefore, necessary to provide a per-rack RM to handle all request/communication for NMs and AMs within the local rack with the global RM only assigning application demands to each of the rack unit resource manager (RU_RM) and monitoring the liveliness of each of these units.

The research objectives are:

1 To parallelise the global control of the RM in YARN framework by providing another layer called rack unit resource manager (RU_RM) layer. This will allow compute nodes on each rack to be controlled by their corresponding rack unit RM instead of a single RM controlling all the compute nodes in the cluster.

2 Carry out a performance evaluation between the developed model and YARN using yarn scheduler load simulator (SLS).

Though the simulator is for testing scheduler performance, it exercises the real YARN RM by simulating NMs and AMs through handling and dispatching NM/app masters heartbeat events within the same JVM. The work will alter the architecture of the simulator to accommodate several RMs which can be used at the rack unit RM layer of the developed model. NMs/app masters together with their schedulers will be re-usable components and will not be re-designed.

2 Related work

Big data has been a challenge for over four decades, with the term changing with advances in information technology. In the '70s, data sizes in megabytes were referred to as 'big'. At a point, data size grew and was measured in gigabytes, then terabytes and petabytes. Today, big data are in zettabytes and yottabytes (Vinayak et al., 2012). Multicore systems were among the early attempts to solve the problem of big data. Machines were made to have dozens of processing cores but with only one disk (Bekkerman et al., 2011). Multicore with their multithread operating systems allows a task to be broken down into smaller units called threads. Threads are then executed concurrently on different CPU cores of the machine (Dilpreet and Chandan, 2014). Peer-to-peer architecture was another approach introduced to overcome the problem of massive data (Steinmetz and Wehrles, 2005). Machines were connected in a decentralised and distributed manner with message passing interface (MPI) serving as a connection protocol within peers (Ripeanu, 2001). With the growing popularity of cloud computing and continuous increase in the volume of data, multicore and peer-to-peer systems became unsuitable for dynamic computations over large amounts of data. The computational capacity today has exceeded the capabilities of conventional processing tools hence, the era of Google file system and MapReduce by Google; and subsequently, the release of classic Hadoop.

Some systems have recognised limitations in Hadoop architecture and have provided alternative models to these limitations. Some of the efforts which closely resemble YARN are COSMOS (Chaiken et al., 2008), Mesos (Hindman et al., 2011), Corona (Facebook, 2012) and Omega (Schwarzkopf et al., 2013) for Microsoft, Twitter, Facebook, and Google respectively. Though these systems share a common inspiration of high-level goals of improving scalability, latency and programming model flexibility, they all have their architectural differences. These differences are most times in diverse design priorities and historical contexts.

COSMOS architectural framework closely resembles that of YARN. The main objective of the framework is to offer availability, reliability, scalability, and performance through its three basic components – COSMOS storage, COSMOS environment, and structured computations optimised for parallel execution (SCOPE). The framework has a job manager (JM), which is the runtime component of the execution engine. It is a central and coordinating daemon for all processing vertices with the application (Chaiken et al., 2008). With multiple applications/ frameworks, COSMOS framework will find it significantly difficult to handle jobs. Mesos architectural design implements an offered-based RM (Hindman et al., 2011), while YARN has a request-based RM (Vinod et al., 2013). Mesos leverages a pool of central schedulers just like the type obtained in classic Hadoop (Zikopoulos and Eaton, 2011) but, YARN uses a per-job intra-framework scheduler which allows AM to request for resources depending on the criteria which includes location, CPU and memory demand (Vinod et al., 2013). Corona is an open-source scheduling framework with a cluster manager responsible for tracking of nodes and free resources in the cluster (Shouvik and Daniel, 2013). Each job has a dedicated JobTracker in this framework. The cluster manager only needs to push resource grants to JabTracker upon request for task execution (Shouvik and Daniel, 2013). The approach used by Corona is a push-based approach, which is different from the heartbeat-based control-plane framework approach in YARN and other frameworks. Though latency/scalability trade-off of these two frameworks deserves a detailed comparison, heartbeat communication protocol negotiates and monitors the availability of a resource in a cluster. It is intended to indicate the health of a machine hence; consideration between overload in YARN due to constant heartbeat between RM and other components and efficient fault tolerance in Corona since it is push-based will have to be looked at. Corona framework also is not very efficient for multiple applications when compared to the Hadoop YARN ecosystem. Omega architectural design geared towards distributed, multi-level scheduling which reflects a greater focus on scalability. It is, however hard to enforce global properties such as capacity/fairness/deadlines on this system (Schwarzkopf et al., 2013).

Other resource allocation systems that attempt to place jobs for optimal performance in big data clusters are Firmament (Gog et al., 2016), h-drf (Bhattacharya et al., 2013), Sparrow (Outerhout et al., 2013), Matrix (Wang et al., 2013), Mercury (Konstantinos et al., 2015), Quarsar (Delimitrou and Kozyrakis, 2014), Awan (Albert et al., 2016) and Justice (Dimopoulos et al., 2017). Most of these systems provide efficient scheduling algorithms but still maintain a single RM in the cluster, which is a bottleneck for scalability (ability to scale-up cluster). Firmament architectural design still maintains a centralised scheduler that can make high-quality placement when scheduling tasks in cluster by continuously rescheduling all tasks

through a min-cost max-flow (MCMF) optimisation. The system was tested using Google workload trace from 12,500 machines and it showed improved placement latency by 20× over Quincy (Gog et al., 2016). Firmament still maintains a central scheduler as YARN model. With increase applications/jobs requesting resources, a single resource allocator will be overwhelmed. H-drf proposed by Bhattacharya et al. (2013) is a hierarchical scheduling system for diverse datacentre workloads in Hadoop. Bhattacharya et al. (2013) maintained that most datacentres exhibit diverse workload with mixed jobs which are sometimes CPU-intensive, memory-intensive or I/O intensive. The system, therefore, uses dominant resource fairness (drf) for job placement. This is a good approach since jobs are allocated based on the type of resources they need. The system, however, did not take into consideration data locality. Outerhout et al. (2013) proposed a distributed, low latency scheduling framework that demonstrates a decentralised, randomised sampling approach for near-optimal performance while avoiding throughput and availability limitations of centralised design. Outerhout et al. (2013) presented sparrow; a stateless distributed scheduler that adapts the power of two choices load balancing technique to the domain of parallel task scheduling. The choices require scheduling each task by probing two random servers and placing the task on the server that is less busy or has fewer queued tasks (Outerhout et al., 2013). Sparrow focused mainly on fine-grained task scheduling for low latency applications. The framework provides task scheduling which is complementary to the functionality provided by cluster managers. Instead of launching a new task, the framework assumes that a long-running execution process is already running on each compute node for each framework hence; it only sends a short task description when a task is launched (Outerhout et al., 2013). The framework makes approximations when scheduling tasks thereby trading off many of the complex features supported by a sophisticated, centralised scheduler to provide higher scheduling throughput and lower latency. This framework does not support gang scheduling typically implemented by bin packing algorithm which searches for reserved time splits on which an entire job can be run. Because Sparrow queues tasks on several machines, it lacks a central point from which to perform bin packing hence, deadlocks between multiple tasks that require gang scheduling may occur. Currently, this framework only supports FIFO order, adding other query-level scheduling policies may improve end-to-end query performance of the framework. It is also important that when a compute node fails, all schedulers with outstanding requests at that node be informed. A centralised state that relies on heartbeat protocol to maintain a list of nodes that are alive may be needed in this framework. Wang et al. (2013) proposed a task execution framework called matrix to overcome Hadoop scaling limitations through distributed task execution. Though matrix was originally developed to schedule executions of data-intensive scientific applications of many-task computing on supercomputers, Wang et al. (2013) saw the need to use the same framework to address scalability issues of Hadoop through decentralising the responsibility of RM. The framework is fully distributed by delegating one scheduler on each compute node (Wang et al., 2013). For each compute node, there is an executor and a key-value store (KVS) server. The scheduler on each of these nodes has the responsibility of managing local resources for optimising load balancing and data-locality (Wang et al., 2013). From the architecture of matrix, it is clear that the framework has a per-node RM (each scheduler maintains a local view of the resources on an individual node). For any framework to have a per-node RM, all data blocks for a single file must be resident on that compute node. Konstantinos et al. (2015) proposed Mercury; a hybrid resource management framework that supports centralised to distributed scheduling on large shared clusters. The placement policy is such that whenever a distributed scheduler needs to place a task on a node, it picks among k nodes with the smallest estimated queuing delay. This allows for optimal performance and improved task throughput (Konstantinos et al., 2015). Data are broken down into blocks and store on several worker nodes in Hadoop cluster. How Mercury ensures central coordination of each of for a single application needs to be looked at. Delimitrou and Kozyrakis (2014) introduced Quasar; a resource-efficient and QoS-aware cluster management system. The sole aim of this system is to provide increase resource utilisation for consistently high application performance (Delimitrou and Kozyrakis, 2014). The system still maintains a single resource allocator which is a bottleneck for scalability. Albert et al. (2016) proposed a framework called Awan; a RM that helps share computing resources across multiple frameworks in an edge cloud environment. The main goal of this system is to provide a general resource management mechanism that will allow each framework to schedule its job with high locality in a geo-distributed environment. To achieve this goal, Awan implements a resource lease abstraction to allocate resources to individual framework schedulers. These schedulers can, in turn, make better scheduling decisions by considering the availability of desirable local resources (Albert et al., 2016). In an attempt to provide a shared-state mechanism where all framework schedulers have global knowledge of all resources in the cluster (both available and non-available resources), resource lease conflicts are bound to occur between schedulers in Awan. Though a mechanism is in place to resolve these conflicts by RM in Awan, an extra overhead will frequently be incurred in running applications in this framework. Justice proposed by Dimopoulos et al. (2017) is a deadline-aware resource allocator that uses deadline information supplied with each job and historical job evaluation logs to implement job placement. The system, however, does not consider jobs without a deadline or job that will not be able to meet the deadline due to resource allocator failure. Panda and Naik (2018) proposed an efficient data replication algorithm for a distributed system. The primary focus of their work is on the redundancy of data at two or more nodes to achieve fault

tolerance and improve cluster performance (Panda and Naik, 2018). The algorithm, called dynamic vote-based data replication (DVDR) is based on dynamic vote assignment among connected nodes which consider all types of faults and re-join failed nodes (Panda and Naik, 2018). Yassir et al. (2019) proposed a related work which is an algorithm that minimises big data movement in a cloud environment. The work focussed on data placement strategies to improve the overall performance of a cluster. Yassir et al. (2019) opined that better data placement and reduction in movement of data through identification of datasets, estimation of load threshold base on processing speed and storage capacity improved overall performance of the cluster. Since data nodes in Hadoop cluster are arranged in racks, careful data placement in these racks to minimise data movement as opined by Yassir et al. (2019) and DVDR by Panda and Naik (2018) need to be looked at for effective replication of data across racks to avoid cross-rack data communication.

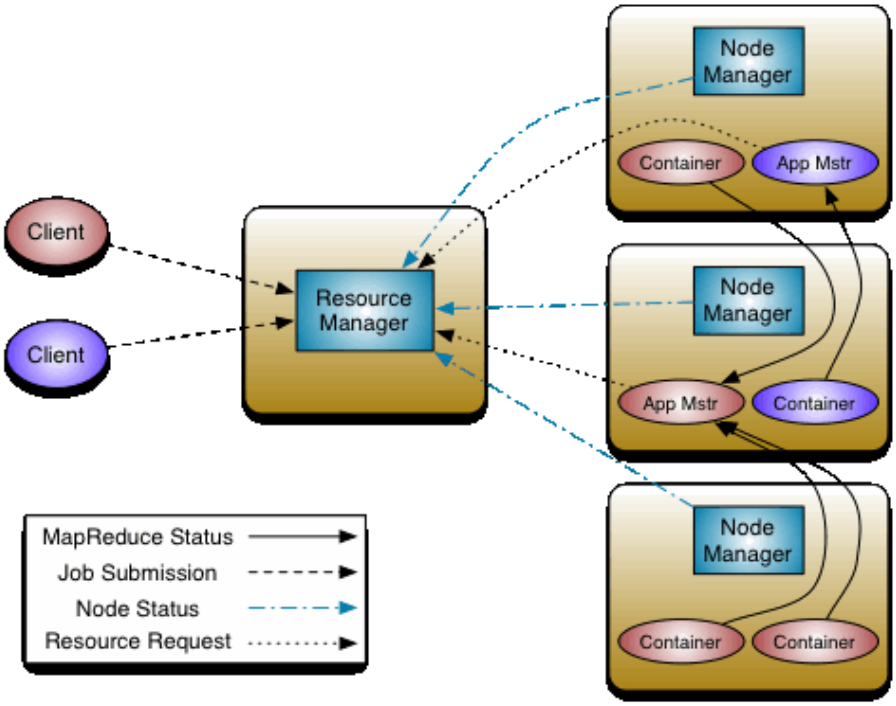
YARN framework stands out among most of the big data analytics due to its ability to run several other frameworks/applications in the cluster. So many other commercial big data analytic frameworks can run on Hadoop YARN which makes it more robust and widely used. Since many other big data analytic tools run on Hadoop YARN, a greater number of application demands is therefore needed which leads to higher resource request from AM in the cluster and worker nodes monitoring through heartbeat communication via NM in each of these worker nodes. The responsibility of a single RM to handle all resource requests from AM and worker node status via NM will constitute a bottleneck for the scalability of Hadoop. This paper aims to parallelise the responsibility of the global RM in YARN by having another layer called rack unit resource managers (RU_RM) responsible for resource management of nodes in their corresponding rack.

3 Architecture of YARN framework

Hadoop YARN has a centralised RM as shown in Figure 1. Though this architectural design has improved scalability significantly, there are fundamental design issues that cap the scalability of this framework towards extreme scales. Some of these design issues include:

1 *Centralised RM:* RM, which is the core component of the Hadoop framework, offers the functionalities of managing, provisioning and monitoring resources like the CPU, memory and network bandwidth of compute nodes. These responsibilities obviously, are bottlenecks for the scalability of Hadoop towards extreme scales. It slows down execution since all compute nodes send/receive instructions from a single RM through heartbeat protocol. Once the RM fails, all execution will halt. Although YARN provides RM high availability to protect against a single point of failure, this technique causes computation overhead because the RM needs to update the backup storage frequently.

Figure 1 Hadoop YARN architecture (see online version for colours)



Source: Apache Hadoop (2018)

2 *Hadoop replication factor:* the replication factor in Hadoop framework is such that 2/3 of each block (of a whole file) is replicated into different data nodes across racks in a cluster. Since AM is expected to monitor the execution of a job/application (with its complete number of blocks) in a cluster, AM will need to communicate data nodes with input splits of the corresponding job/application across racks to be able to monitor this execution. Communication across racks will results in higher latency for job execution.

3 *Job completion time:* since only RM coordinates the release of resources for execution of jobs, several AMs polling from RM of this framework during resource request is a bottleneck for the system. It slows down processing, which means that total turnaround time (job completion time) for each job will be high.

4 Architecture of the developed model

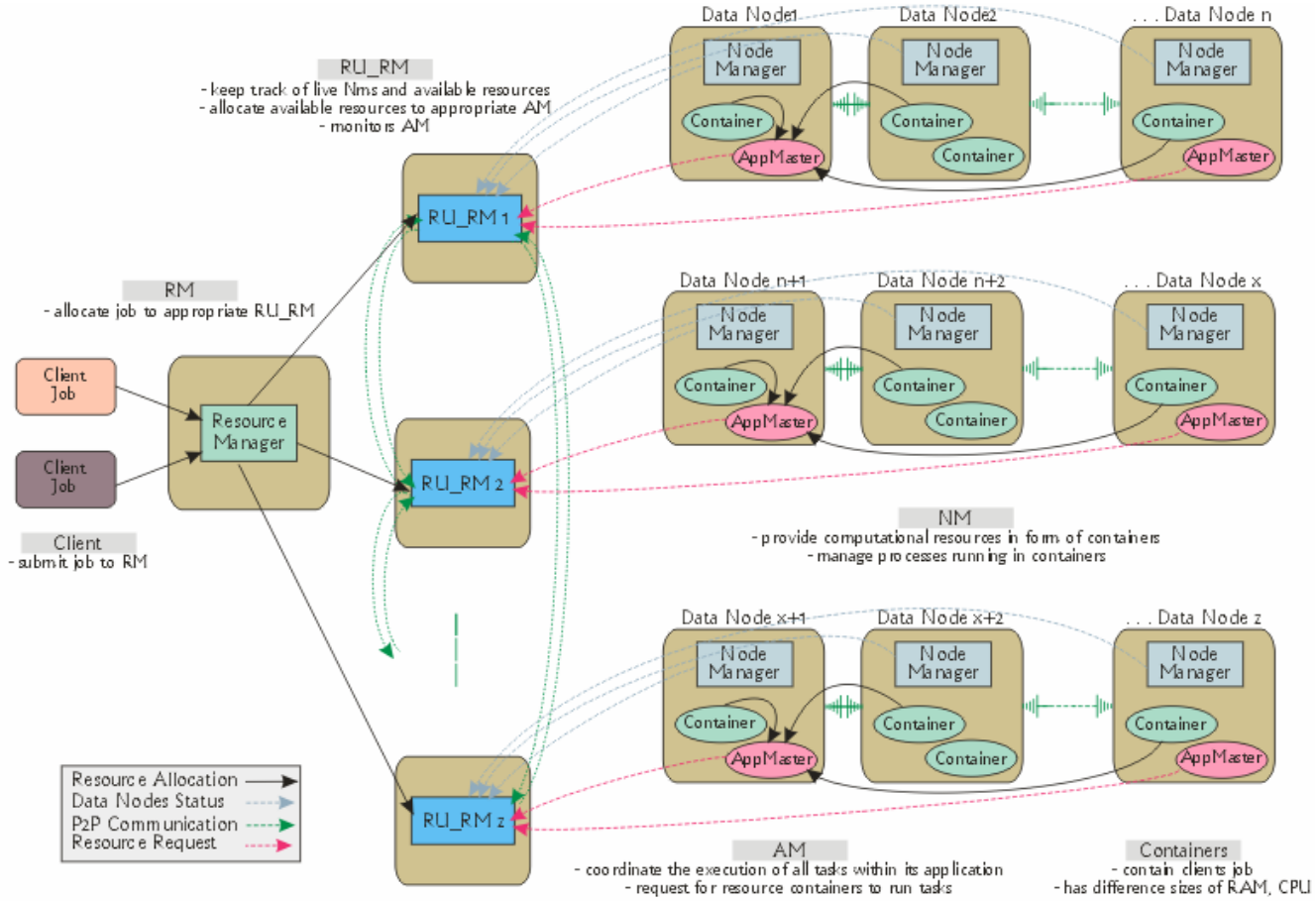
The architecture of the developed model is shown in Figure 2. The main aim of this model is to parallelise the global control of the RM in YARN framework by providing another layer called rack unit resource manager (RU_RM) layer. This layer aims to make compute nodes on each rack to be controlled by their corresponding rack unit RM instead of a single RM controlling all the compute nodes in the network. We believe that this will help improve response and turnaround time for each job/application and will eliminate single point of failure which makes jobs halt in the existing model.

Central RM obtains information on which rack contains 2/3 of the total number of blocks for each job from the name node and pushes a job to the corresponding RU_RM. RU_RM decides on resource lease requests from the AM in the rack.

5 Hypothetical test

To justify the working methodology of the developed model, a hypothetical evaluation is carried out to analyse task response time for both the developed and existing YARN models. With another level of abstraction introduced below the global RM, this test is needed to determine task response time at both central RM and rack unit RM layer of the developed model compare to task response time at the global RM in YARN model.

Figure 2 Architecture of the developed model (see online version for colours)



Let us assume to have three jobs (applications) to be processed and that, each instruction cycle in executing any of these jobs takes 0.01ns (assume that three jobs are of the same size). For each job, therefore, the first five instruction cycles in the existing framework will be 0.05ns (as obtained from Algorithm 1). Since there is only one RM, the last six instructions will require RM communicating with NM to launch containers and with AM for resource requests. Since no more than one instruction can be given at a time, it means that RM will interleave these instructions between the three jobs (applications). Assume by intuition, that resource lease from RM to AM takes 3 ns and the interleaving of processes follows FIFO order. It means that *Job 1* gets resources immediately hence, the delay time is zero (0). *Job 2* will get access to a resource at time 3 ns while *Job 3* at time 6 ns. The overall time it will take to process the three jobs will be as follows:

$$Job\ 1 = (0.01\ ns \times 5) + 0\ ns = 0.05\ ns$$

$$Job\ 2 = (0.01\ ns \times 5) + 3\ ns = 3.05\ ns$$

$$Job\ 3 = (0.01\ ns \times 5) + 6\ ns = 6.05\ ns$$

Total task response time needed to process the three jobs = 9.15 ns.

Algorithm 1 Job execution in YARN

a. Job submission phase

Step 1: *The job gets submitted to the job client.*

Step 2: *The job client request for a new job id.*

Step 3: *The job client then checks if the output directory has been created. After verifying this, it copies the job resources to the HDFS.*

Step 4: *The job client then submits the job to the resource manager.*

b. Job initialisation phase

Remember that, resource manager has two units – resource scheduler and application manager. The scheduler schedules and allocates resources while application manager monitors the status and process of the job.

Step 5: *As soon as the scheduler picks a job, it contacts the node manager to allocate a container and launch application master for the job.*

Step 6: *Application master creates an object for the job. This is done for bookkeeping purposes and task management.*

Step 7: *The application master retrieves input splits from HDFS and creates 1 map per split. The application master at this point decides how to execute the job. If the job is a small task, the application master runs the job in its JVM to avoid unnecessary overhead. These types of tasks are called Uber tasks in Hadoop framework.*

c. Task assignment phase

Step 8: *If the job is large, the application master requests the resource manager to allocate the computing resources needed. Scheduler at this point knows where the resources are located. It gathers this information from the heartbeat it gets from each worker node. It uses this information to consider data locality while assigning a task. The scheduler tries as much as possible to assign a task to where the data are located. If this is not possible, it assigns the task to another node within the cluster.*

d. Task execution phase

Step 9: *Application master contacts the node manager assigned to execute the task, to start a container. The node manager then launches the YARN child.*

YARN child is a Java program that has the main class 'YarnChild'. It runs a separate JVM to isolate user code from a long-running system.

Step 10: *YarnChild retrieves all job resources from the HDFS.*

Step 11: *YarnChild now runs the map and reduce tasks.*

e. Progress and update phase

In this phase, YarnChild sends the progress report every 3 seconds to the application master. Application master in turn aggregates and sends an update directly to the job client.

f. Job completion phase

Application master and task containers clean up their working state.

With the developed model, the first seven instruction cycles will be 0.07 ns (as obtained in Algorithm 2). Since each RU_RM node executes just one job at a time, the last six instructions therefore are carried out at the same time on different RU_RM node. Therefore, if it takes 3 ns for resource lease in the existing system; it will take *1/3 ns of 5 steps* for resource lease in the new model. Hence, the process time for the three jobs will be as follows:

$$Job\ 1 = (0.01\ ns \times 7) + 1/3\ ns\ of\ 5\ on\ RU\_RM1$$
$$= 0.07\ ns + 1.67\ ns = 1.74\ ns$$

$$Job\ 2 = (0.01\ ns \times 7) + 1/3\ ns\ of\ 5\ on\ RU\_RM2$$
$$= 0.07\ ns + 1.67\ ns = 1.74\ ns$$

$$Job\ 3 = (0.01\ ns \times 7) + 1/3\ ns\ of\ 5\ on\ RU\_RM3$$
$$= 0.07\ ns + 1.67\ ns = 1.74\ ns$$

Total task response time needed to process these three jobs = 5.22 ns.

Algorithm 2 Job execution in the developed model

a. Job submission phase

Step 1: *The job gets submitted to the job client.*

Step 2: *The job client request for a new job id.*

Step 3: *The job client then checks if the output directory has been created. After verifying this, it copies the job resources to the HDFS.*

Step 4: *The job client then submits the job to the resource manager.*

b. *Job initialisation phase*

Step 5: *Resource manager gets input splits for the said job.*

Step 6: *With the information in Step 5, the resource manager schedules appropriate rack unit resource manager (RU_RM) with 2/3 of the input split to execute the job.*

Step 7: *The scheduler at the RU_RM picks the job and contacts the appropriate node manager to launch application master for the job.*

Step 8: *Application master creates an object for the job. This is done for bookkeeping purposes and task management. The application master creates 1 map per split from each input split on the data node. The application master at this point decides how to execute the job. If the job is a small task, the application master runs the job in its JVM to avoid unnecessary overhead.*

c. *Task assignment phase*

Step 9: *If the job is large, the application master requests the rack unit resource manager to allocate the computing resources needed (container). Scheduler at this point knows where the resources are located. It gathers this information from the heartbeat it gets from each worker node in the rack. It uses this information to consider data locality while assigning a task. The scheduler tries as much as possible to assign a task to where the data are located. If this is not possible, it assigns the task to another node within the rack.*

d. *Task execution phase*

Step 10: *The rack unit resource manager through the appropriate node manager launches the YARN child.*

Step 11: *YarnChild retrieves all job resources from the HDFS.*

Step 12: *YarnChild now runs the map and reduce tasks.*

e. *Progress and update phase*

In this phase, YarnChild sends the progress report every 3 seconds to the application master. Application master in turn aggregates and sends an update directly to the job client.

f. *Job completion phase*

Application master and task containers clean up their working state.

6 Experimental setup and results

This section evaluates the developed model by comparing it with the existing model in processing typical Hadoop workloads like sort, WordCount, TeraSort, PageRank, naïve Bayes, and K-means. The experimental cluster for this evaluation consists of four systems. One of these four systems is designed to be both master and a slave node while the other three are slave nodes. We have four nodes with each node interconnected by 10 G-Ethernet. Each node has two Intel Xeon CPU E5-2670 running at 2.60 GHz. Each CPU has eight cores (each core has two threads: hyper-threading). The memory size for each node is 64 GB. CentOS 6.7 operating system and Java 1.8.0 were used for the cluster. YARN SLS's architecture was altered to accommodate several RMs used as rack unit RMs for the developed model. NMs and AMs together with their schedulers were re-usable components for the developed model. We set HDFS block replication to be 3 and block size to 128 MB.

6.1 Definition of metrics

We define two metrics to evaluate the performance of YARN and improved YARN. The two metrics are efficiency and average-delay ratio.

6.1.1 Efficiency

Efficiency in this work represents the percentage of the total ideal finished time (Total-T_{ideal}) to the actual finished time (T_{actual}) of a task. T_{ideal} is run time obtained by running a single data block (without overheads). A single data block was set as an Uber task, which was processed by AM in its JVM to avoid unnecessary overhead. T_{actual} is the total running time (with overheads) obtained by executing workloads on existing and developed models.

This metric helps quantify the average cluster utilisation of the developed and existing model. This is represented by equation (1).

$$Efficiency = \left(Total - T_{ideal} / T_{actual}\right) * 100\% \tag{1}$$

$$Total - T_{ideal} = number\ of\ blocks\ for\ each\ dataset * T_{ideal} \tag{2}$$

6.1.2 Average task-delay ratio

Average task-delay ratio (r_{td}) for this work is computed as the normalised difference between ideal task finished time (T_{itf}) and average actual task finished time (T_{atf}).

This metric measures how fast the models can respond from a task's perspective. This is represented by equation (3).

$$r_{td} = \left(T_{atf} - T_{itf}\right) / T_{itf} \tag{3}$$

where

$$T_{atf} = T_{actual} / number\ of\ blocks \tag{4}$$

$$T_{itf} = T_{ideal} \tag{5}$$

6.2 Benchmarks

We use six benchmarks to compare our results. The benchmarks are sort, WordCount, TeraSort, PageRank, naïve Bayes, and K-means. Sort workload is used to sort input text files by key. WordCount reads a text file and count the number of occurrences of each word. TeraSort has three applications – TeraGen used to generate input data, TeraSort for sorting input data and TeraValidate for checking the output. PageRank ranks web pages in its search engines. Naïve Bayes is a machine learning benchmark based on Bayes' theorem with independence assumptions between features to categorise text. K-means is used to group items into k clusters (Huang et al., 2011).

These six benchmarks cover HiBench benchmark suites – micro benchmark (sort, WordCount, and TeraSort), web search (PageRank) and machine learning (naïve Bayes and K-means clustering). We use the Hadoop benchmark and data generator provided in HiBench benchmark suite 4.0 from https://github.com/intel-hadoop/HiBench.

6.3 Results and analysis

We present the results of experiments and analysis for benchmarks described above. To compare the performance of the developed and existing models, we record the running time for each workload operation. The running time was recorded in seconds.

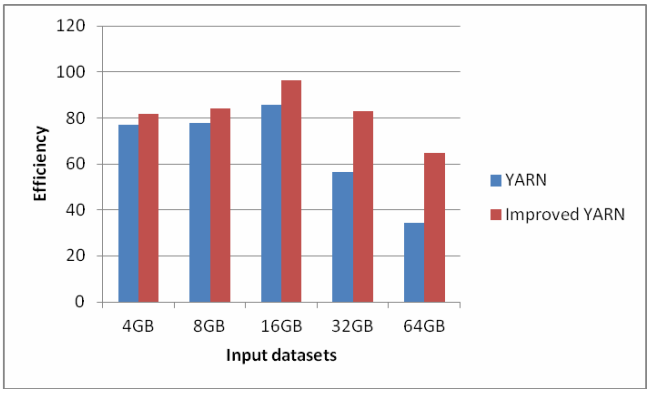
6.3.1 Sort

Table 1 represents running time (T_{actual}) performance comparison for sort on YARN and improved YARN model.

Table 1 Sort running time (seconds)

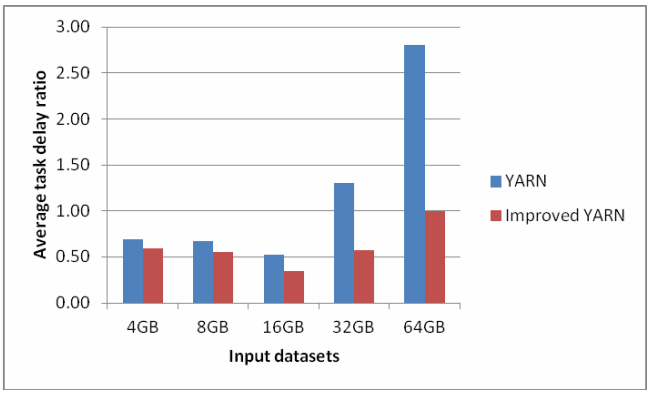
Input	*4 GB*	*8 GB*	*16 GB*	*32 GB*	*64 GB*
YARN	54	107	194	589	1,947
Improved YARN	51	99	173	402	1,025

Figure 3 Efficiency from running sort workload (see online version for colours)



The input sizes for these datasets range from 4 GB to 64 GB. To obtain ideal runtime (T_{ideal}) for this workload, we ran a single block size of 128 MB which gave an average of 1.3 seconds after three attempts. Efficiency and average-delay ratio were calculated from equations (1) and (3) for both YARN and improved YARN models with the chart represented in Figures 3 and 4 respectively.

Figure 4 Average-task delay ratio from running sort workload (see online version for colours)



From Figure 3, improved YARN outperformed the existing YARN model with efficiency 2× better when input size increased to 64 GB. Also, the average-task delay as shown in Figure 4 for the developed model is closely 3× less than the existing model, which also reflected in the overall performance. Both YARN and improved YARN performed poorly as input size increase. This may be as a result of the number of nodes, memory sizes and processor speeds of the nodes used for the experiment.

6.3.2 WordCount

Running time for YARN and improved YARN on WordCount workload is as presented in Table 2.

Table 2 WordCount running time (seconds)

Input	*4 GB*	*8 GB*	*16 GB*	*32 GB*	*64 GB*
YARN	267	509	982	2,023	4,078
Improved YARN	235	413	822	1,649	3,309

The input sizes for these datasets range from 4 GB to 64 GB. To obtain T_{ideal}, we ran a single block size of 128 MB which gave an average of 6.2 seconds after three attempts. Efficiency and average-delay ratio were calculated from equations (1) and (3) for both YARN and improved YARN models with the chart represented in Figures 5 and 6 respectively.

From Figure 5, we see that improved YARN outperformed YARN. As the input dataset increases from 16 GB to 64 GB, YARN's efficiency drops by 3% while improved YARN maintained a nearly consistent performance dropping by less than 1%. The results show that improved YARN is more scalable than YARN in the presence of an increased workload. This is largely due to the distributed architecture in the rack unit RM layer. The average-task delay ratio shows a consistent delay for improved YARN while task-delay in YARN has a

continuous increase in time as the input dataset increases, yielding poor efficiency on cluster utilisation. This is due to all task pulling resource requests from a single RM.

Figure 5 Efficiency from running WordCount workload (see online version for colours)

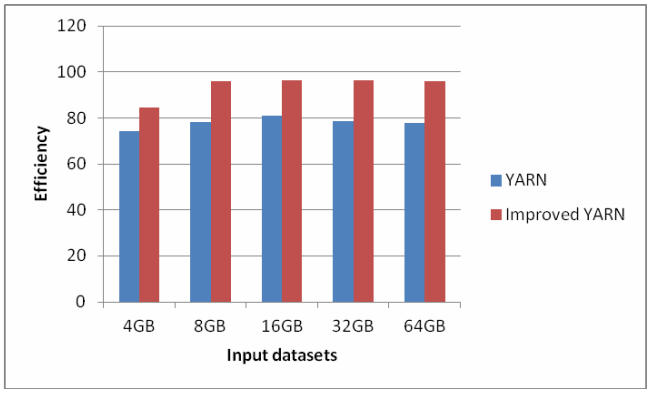
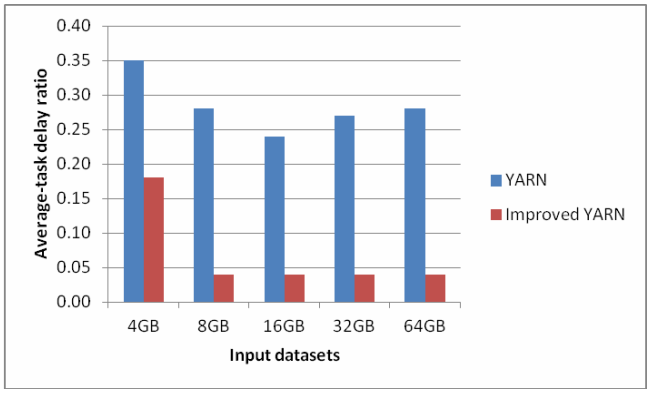


Figure 6 Average-task delay ratio from running WordCount workload (see online version for colours)



6.3.3 TeraSort

We ran a TeraSort test for assessing YARN and improved YARN performance using input datasets of sizes ranging from 8 GB to 128 GB. First, we generate these data using TeraGen and then used TeraSort to sort the data. Finally, we validate the sorted results using TeraValidate. Table 3 shows the runtime (T_{actual}) performance of TeraSort from each input size.

Table 3 TeraSort running time (seconds)

Input	*8 GB*	*16 GB*	*32 GB*	*64 GB*	*128 GB*
YARN	85	156	572	1,219	3,334
Improved YARN	84	151	566	1,134	3,013

We ran a single block comprising 128 MB with three iterations. The average ideal runtime (T_{ideal}) obtained for a single block was 1.1 seconds. Figures 7 and 8 show the efficiency and average-task delay ratio for both YARN and improved YARN models.

Figure 7 Efficiency from running TeraSort workload (see online version for colours)

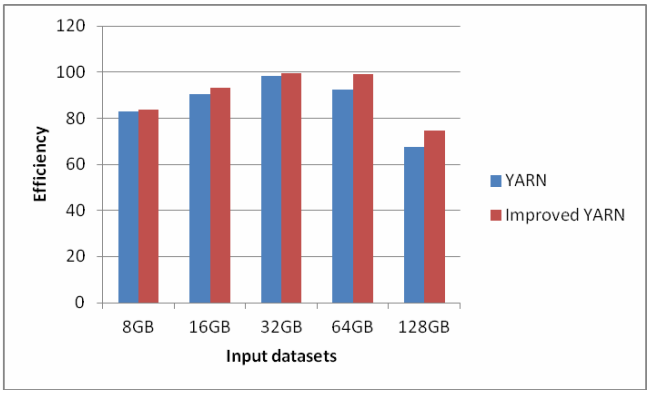
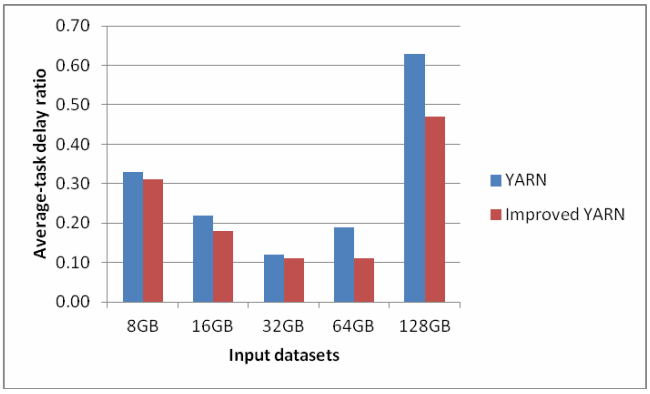


Figure 8 Average-task delay ratio from running TeraSort workload (see online version for colours)



We observed that YARN achieved performance close to improved YARN. However, there is still a significant difference with improved YARN outperforming YARN as input datasets increases. Also, task delay for both models dropped between input data sizes of 16 GB and 64 GB with a significant rise in the existing model when input data size increased to 128 GB. This significant increase shows that with more datasets, improved YARN will be more scalable than YARN.

6.3.4 PageRank

Table 4 represents runtime performance comparison for PageRank on YARN and improved YARN.

Table 4 PageRank running time (seconds)

Input	*2 M*	*4 M*	*8 M*	*16 M*	*30 M*
YARN	405	715	1,401	2,852	5,562
Improved YARN	396	675	1,297	2,614	5,001

The input datasets for this experiment as shown in Table 4 ranges from 2 million pages to 30 million pages. The sizes of these input datasets are from 1.3 GB to 19.7 GB. To obtain ideal runtime for this workload, we ran a single block which is approximately 200 thousand pages

with three iterations. Average ideal runtime (T_{ideal}) of 30.1 seconds was obtained. Efficiency and average-task delay ratio for both existing and developed models are shown in Figures 9 and 10.

Figure 9 Efficiency from PageRank workload (see online version for colours)

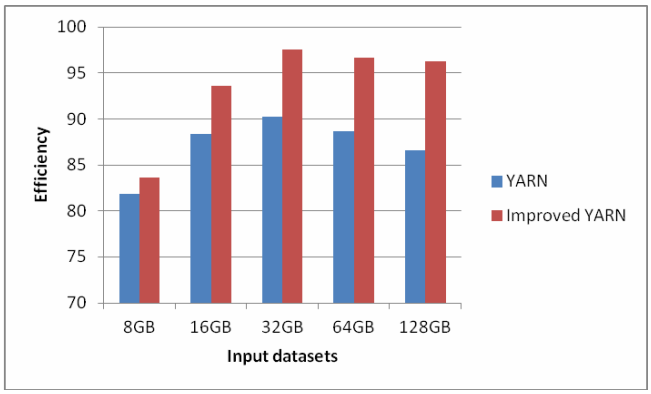
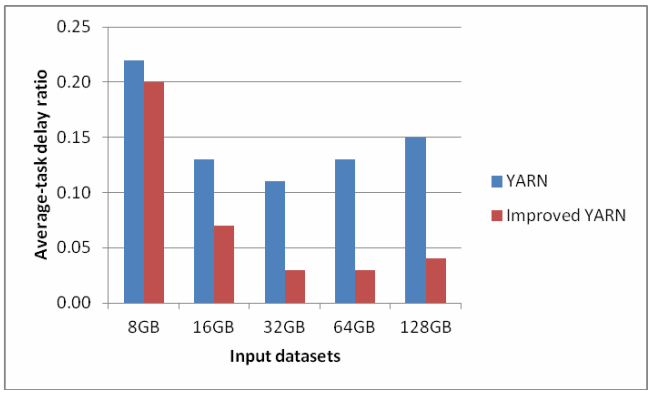


Figure 10 Average-task delay ratio from PageRank workload (see online version for colours)



Input datasets of 2 M and 4 M show no significant differences in the efficiency of YARN and improved YARN model. With an increase in input datasets, however, improved YARN performed better than YARN. With an increase in input datasets also, task delay for improved YARN is 2× less compared to delay on the YARN model. This is largely due to multiple RMs at rack unit RM layer of improved YARN.

6.3.5 *Naïve Bayes*

Table 5 represents running time (T_{actual}) performance for naïve Bayes between YARN and improved YARN.

Table 5 PageRank running time (seconds)

Input	*100 K*	*200 K*	*400 K*	*800 K*	*1.6 M*
YARN	636	1,289	2,475	4,789	12,725
Improved YARN	608	1,211	2,293	4,462	10,322

The input datasets are from 100 thousand to 1.6 million which ranges from 0.45 GB to 7.5 GB. Ideal run time for a single block of approximately 25 k was 149 seconds. Figures 11 and 12 show the efficiency and average-task delay ratio for YARN and improved YARN.

Figure 11 Efficiency from naïve Bayes workload (see online version for colours)

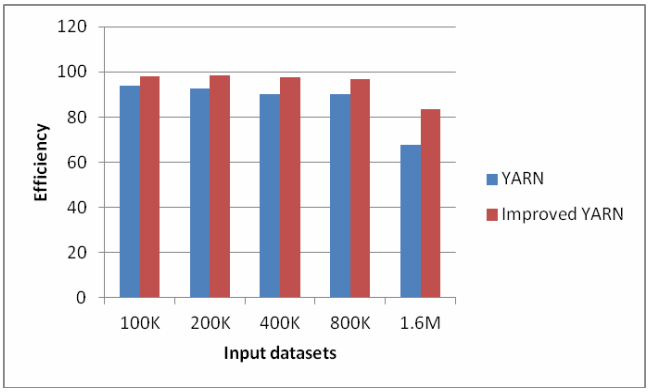
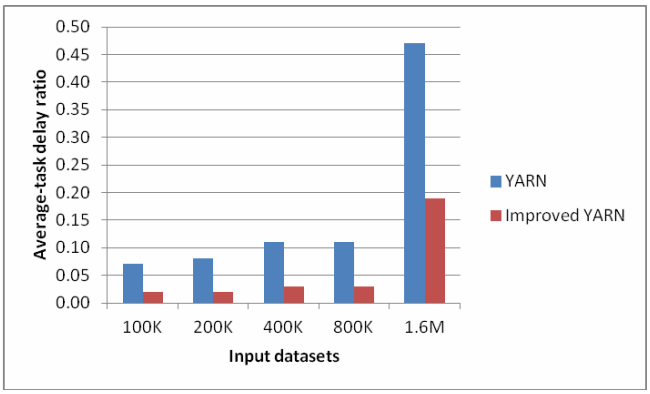


Figure 12 Average-task delay ratio from naïve Bayes workload (see online version for colours)



Improved YARN shows consistency in cluster utilisation only dropping by approximately 2% as input datasets get bigger. Though YARN also shows the same consistency, performance dropped significantly when the input dataset became 1.6 M.

6.3.6 *K-means*

Runtime performance for YARN and improved YARN using K-means workload is shown in Table 6. Input datasets for this experiment ranges from 50 M to 800 M records with sizes ranging from 10 GB to 160 GB. Figures 13 and 14 show the efficiency and average-task delay ratio for YARN and improved YARN models.

Table 6 K-means running time (seconds)

Input	*50 M*	*100 M*	*200 M*	*400 M*	*800 M*
YARN	334	656	1,357	2,755	6,553
Improved YARN	327	643	1,336	2,694	5,426

Figure 13 Efficiency from K-means workload (see online version for colours)

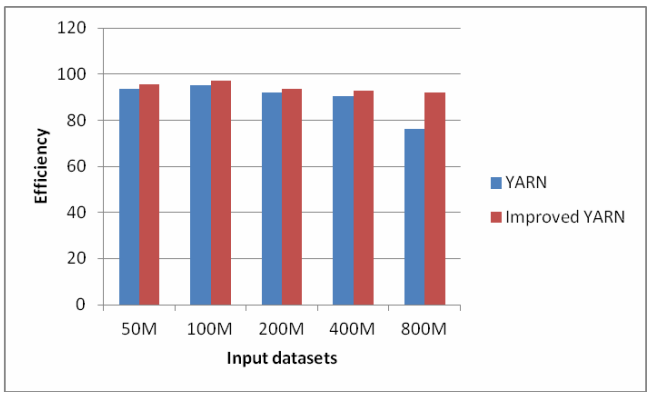
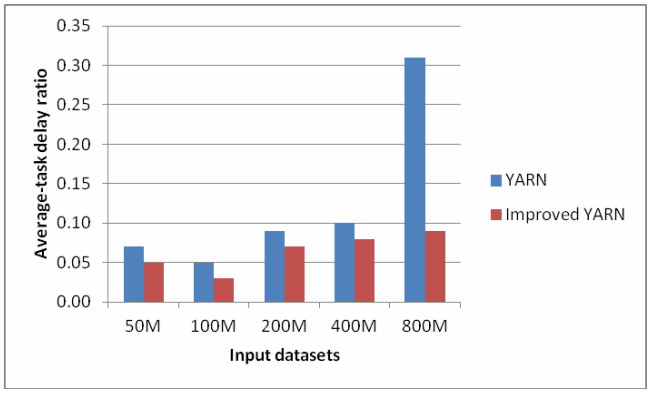


Figure 14 Average-task delay ratio from K-means workload (see online version for colours)



We ran this algorithm using five clusters and obtained an ideal runtime of 3.9 seconds for a single block of approximately 7 M input data size. YARN performed poorly as input datasets increases. Improved YARN shows a much slower decreasing trend and became stable from 200 M to 800 M in terms of efficiency and average-task delay ratio. The increasing trend in YARN and much stable performance of improved YARN is likely to hold for more input datasets. The performance of improved YARN is in the distributed architecture of RMs in the cluster.

7 Conclusions

A model of improved scalable resource management system for Hadoop YARN is an improvement over the existing Hadoop YARN model. The finished time of workload operation in the log file for YARN and improved YARN was used as a basis for comparing the models. Results obtained from computation of efficiency and average task-delay ratio showed that as file size increases, improved YARN performs better than the existing YARN model. Since Hadoop framework is intended for massive data, it implies that improved YARN promises to proffer better resource management for a scalable Hadoop framework.

Future work will look at providing efficient fault tolerance mechanisms at the Rack_Unit RM layer to prevent any fault that may lead to computation issues for data nodes on any of the rack. We will also look at how this solution can be implemented in other big data resource allocation systems.

References

Aboudi, N.E. and Benhlima, L. (2017) 'Parallel and distributed population based feature selection framework for health monitoring', *International Journal of Cloud Applications and Computing*, Vol. 7, No. 1, pp.57–71.

Albert, J., Abhishek, C. and Jon, W. (2016) 'Awan: locality-aware resource manager for geo-distributed data-intensive applications', *IEEE International Conference on Cloud Engineering (IC2E)*, IEEE Computer Society, Berlin, Germany.

Apache Hadoop (2018) *Apache Hadoop YARN* [online] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html (accessed 19 November 2018).

Bekkerman, R., Bilenko, M. and Langford, J. (2011) *Scaling Up Machine Learning: Parallel and Distributed Approaches*, Cambridge University Press, England.

Bhattacharya, A.A., Culler, D., Freedman, E., Ghodsi, A., Shenker, S. and Stoica, I. (2013) 'Hierarchical scheduling for diverse datacentre workloads', *Proceedings of the 4th Annual Symposium on Cloud Computing*, ACM, Santa Clara, California.

Bialecki, A., Cafarella, M., Cutting, D. and O'Malley, O. (2005) *Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware* [online] http://lucene.apache.org/hadoop/ (accessed 6 June 2015).

Chaiken, R., Jenkins, B., Larson, P.A., Ramsey, B., Shakib, D., Weaver, S. and Zhou, J. (2008) 'Scope: easy and efficient parallel processing of massive data sets', *Proceedings of the VLDB Endowment*, Vol. 1, No. 2, pp.1265–1276.

Dean, J. and Ghemawat, S. (2004) 'MapReduce: simplified data processing on large clusters', *OSDI '04: 6th Symposium on Operating Systems Design and Implementation*, USENIX Association, Berkeley, CA.

Delimitrou, C. and Kozyrakis, C. (2014) 'Quarsar: resource-efficient and QoS-aware cluster management', *ASPLOS '14*, ACM, Lake City, Utah, USA.

Dilpreet, S. and Chandan, K.R. (2014) 'A survey on platforms for big data analytics', *Journal of Big Data: A Springer Open Journal* [online] https://journalofbigdata.springeropen.com/articles/10.1186/s40537-014-0008-6 (accessed 24 June 2016).

Dimopoulos, S., Krintz, C. and Wolski, R. (2017) 'Justice: a deadline-aware fair-share resource allocator for implementing multi-analytics', *Proceedings of the IEEE International Conference on Cluster Computing*, pp.233–244.

Dominique, A.H. (2015) 'Hadoop design, architecture and MapReduce performance', *DH Technologies* [online] http://www.dhtusa.com (accessed 10 March 2015).

Facebook (2012) *Under the Hood: Scheduling MapReduce Jobs More Efficiently with Corona* [online] http://on.fb.me/TxUsYN (accessed 21 October 2015).

Fu, Z., Wu, Z., Li, H., Li, Y., Wu, M., Chen, X., Ye, X., Yu, B. and Hu, X. (2019) 'GeaBase: a high-performance distributed graph database for industry-scale applications', *International Journal of High Performance Computing and Networking*, Vol. 15, Nos. 1–2, pp.12–21.

Gog, I., Gleave, A. and Watson, R.N.M. (2016) 'Firmament: fast, centralized cluster scheduling at scale', *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association, Savannah, GA, USA.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S. and Stoica, I. (2011) 'Mesos: a platform for fine-grained resource sharing in the data center', in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, Berkeley, CA, USA.

Huang, S., Huang, J., Dai, J., Xie, T. and Huang, B. (2011) 'The HiBench benchment suit: characterization of the MapReduce-based data analysis', in Agrawl, D., Candan, K.S. and Li, W.S. (Eds.): *New rontiers in Information and Software as Servces. Lecture notes in Business Information Processing*, Vol 74, Springer, Berlin, Heidelberg.

Konstantin, S., Hairong, K., Sanjay, R. and Robert, C. (2010) 'The Hadoop distributed file system', *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, IEEE, NV, USA.

Konstantinos, K., Rao, S., Curino, C., Douglas, C., Chaliparambil, K., Fumarola, G.M., Heddaya, S., Mamakrishnan, R. and Sakalanaga, S. (2015) 'Mercury: hybrid centralized and distributed scheduling in large shared clusters', *Proceedings of USENIX Annual Technical Conference*, USENIX Association, Santa Clara, CA, USA.

Outerhout, K., Patrick, W., Matei, Z. and Ion, S. (2013) 'Sparrow: distributed, low latency scheduling', Hertz Foundation Fellowship, ACM, Pennsylvania, USA, DOI: dx.doi.org/ 10.1145/2517349.2522716.

Panda, S.K. and Naik, S. (2018) 'An efficient data replication algorithm for distributed systems', *International Journal of Cloud Applications and Computing*, Vol. 8, No. 3, pp.60–77.

Ripeanu, M. (2001) 'Peer-to-peer architecture case study: Gnutella network', *Proceedings of First International Conference on Peer-to-Peer Computing*, IEEE, Linkoping, Sweden.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M. and Wilkes, J. (2013) 'Omega: flexible, scalable schedulers for large compute clusters', in *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13*, ACM, New York, NY, USA.

Sergio, C.G. (2015) *What about Big Data?*, A Project Carried Out at Computer Science and Engineering Department of the Open University of Catalonia.

Shouvik, B. and Daniel, A.M. (2013) 'The anatomy of MapReduce jobs, scheduling and performance challenges', *Proceedings of the 2013 Conference of the Computer Measurement Group*, Semantic Scholar, San Diego, CA.

Steinmetz, R. and Wehrles, K. (2005) *Peer-to-Peer Systems and Applications*, Springer, Berlin, Heidelberg.

Vinayak, R.B., Michael, J.C. and Chan, L. (2012) 'Big data platforms: what's next?', *ACM Transactions on Accessible Computing*, Vol. 9, No. 1, pp.44–49.

Vinod, K.V., Arun, C.M., Chris, D., Sharad, A., Mahadev, K., Robert, E., Thomas, G., Jason, L., Hitesh, S., Siddaharth, S., Bikas, S., Carlo, C., Owen, O.M., Sanjay, R., Benjamin, R. and Eric, B. (2013) 'Apache Hadoop YARN: yet another resource negotiator', *SOCC '13 Proceedings of the 4th Annual Symposium on Cloud Computing*, ACM, New York, NY [online] http://dx.doi.org/10.1145/2523616.2523633.

Wang, K., Ma, Z. and Raicu, I. (2013) 'Modelling many-task computing workloads on a Petaflop IBM BlueGene/P supercomputer', *IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum*, IEEE Computer Society, Massachusetts, USA.

Wu, J., Guo, S., Li, J. and Zeng, D. (2006) 'Big data meet green challenges: big data toward green applications', *IEEE Systems Journal*, Vol. 10, No. 3, pp.888–900.

Yassir, S., Zbakh, M. and Claude, T. (2019) 'Graph-based model and algorithm for minimizing dig data movement in a cloud environment', *International Journal of High Performance Computing and Networking*, Vol. 14, No. 3, pp.365–375.

Zikopoulos, P. and Eaton, C. (2011) *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*, 1st ed., McGraw-Hill, Osbome Media [online] https://dl.acm. org/doi/book/10.5555/2132803 (accessed 30 January 2012).