Self Adaptive Hadoop Scheduler for Heterogeneous Resources

Amr M. Elkholy, Elsayed A.H. Sallam
Computers and Automatic Control Dept.
Faculty of Engineering
Tanta University
Tanta, Egypt
Email: {amr.elkholy, sallam}@f-eng.tanta.edu.eg

***Abstract*—Nowadays, Hadoop is a widely used framework for processing large data. Hadoop scheduler is a critical element which has a big effect on Hadoop performance. Finding a dynamic scheduler which adapts to different nodes computing capabilities and the same node performance is a challenging problem. Most of the current Hadoop schedulers consider the homogeneity of the resources on which Hadoop is running and assign each node in the cluster a fixed capacity over the run time, neglecting the different nodes computing capabilities and the performance of each node over the run time. This causes under/over utilization of resources, poor performance and longer run time. So, we propose a dynamic Hadoop scheduler which adapts to the performance and the computing capabilities of each node separately. The proposed scheduler controls the capacity of each node which represented by the number of tasks that can be processed concurrently at a time. The scheduler extends/shrinks the capacity of each node depending on its available resources and performance over the run time. Our scheduler is implemented on Hadoop and compared by the Hadoop Fair Scheduler. The experimental results show that our scheduler has achieved less average completion time and higher resources utilization.**

***Keywords*—Hadoop, MapReduce, Scheduling.**

I. Introduction

Big data is a common research topic in recent years due to the vast amount of data generated every day. Finding efficient tools for processing, analyzing and utilizing big data is a challenging problem. MapReduce [1] is one of the efficient tools for processing vast amount of data in parallel. Its simplicity and flexibility makes MapReduce a common tool for processing big data used in many enterprises. Hadoop [2] is one of the most popular implementations of MapReduce which has been used by Yahoo and Facebook for large data parallel distributed processing. Hadoop scheduler is a critical element which has a big effect on the Hadoop performance. To the best of our knowledge most of the current Hadoop schedulers don't consider the different nodes computing capabilities on which Hadoop is running and the node performance over the run time, and give each node a fixed number of slots. This assumption makes the capacity of the resource static over the run time neglecting the machine specifications, performance and tasks heterogeneity. From our experimental study of Hadoop, we found that assumption leads to under or over utilization of resources, which then leads to longer run time. So, we propose a new Hadoop scheduler that monitors the resources utilization (CPU, memory and network bandwidth) of each node separately and extends/shrinks the capacity of each node over the run time depending on the resources availability and the needs of running jobs. In our scheduler we don't need any prior execution of the job as in [3]. We implemented our proposed scheduler on Hadoop. The scheduler was evaluated on a heterogeneous set of nodes and the results show that our scheduler achieves higher performance and more resources balanced utilization compared to the Hadoop Fair Scheduler [4]. Beside this introduction, in Section II, we give a MapReduce background and the related work. Our proposed scheduler is described in Section III. In Section IV, scheduler implementation and the experimental results are described. Finally, the paper conclusion and future work are drawn in Section V.

II. Background and Related work

A. MapReduce

MapReduce [1] is a popular programming model for processing large data in parallel. It consists of two main phases, the 1^{st} phase is the map phase which processes a block of the input data, which is in the form of key-value pairs and produces intermediate result also in the form of key-value pairs. The 2^{nd} phase is the reduce phase which combines the intermediate results generated by the map phase and produces the final result. One of the most famous implementations of MapReduce paradigm is Hadoop which will be discussed in the following subsection.

B. Hadoop

Hadoop [2] is an open source implementation of the MapReduce paradigm provided by Apache Software Foundation. Hadoop consists of Hadoop Distributed File System (HDFS) and MapReduce. The Hadoop cluster consists of a master node which runs the JobTracker process and one or more slave nodes each runs a TaskTracker process. The JobTracker is responsible for accepting jobs from users, partitioning input data into blocks, assigning tasks to TaskTrackers and coordinating work across the cluster. On the other hand, the TaskTrackers are responsible for controlling the execution of map and reduce tasks. When submitting a job to Hadoop, the data are partitioned into blocks, for each block there is a map task assigned to a TaskTracker when has an idle map slot which runs the same function on a different block of data. After a TaskTracker finishes the execution of the map task, the intermediate data are transferred to the node which

978-1-4799-6594-6/14/$31.00 ©2014 IEEE

will perform the reduce task. In Hadoop, the submitted job consists of a set of map and reduce tasks. Each resource is represented as a set of map and reduce slots. The map slots for map tasks and the reduce slots for reduce tasks. These resources need to be distributed among users/submitted jobs. So, Hadoop uses a scheduler responsible for distributing the cluster resources among users/submitted jobs. The default scheduler used by Hadoop is FIFO (First In First Out) scheduler which doesn't consider heterogeneity of resources. The Hadoop Fair scheduler [4] considers fairness by sharing cluster resources among users in which a pool is defined for each user. Each pool consists of a number of map and reduce slots, within each pool the slots are distributed among the user submitted jobs. Which tries to achieve fairness among submitted jobs, but still have the problem of not considering the different computing capabilities and the performance of the nodes on which Hadoop is running.

C. Related work

There is a lot of research work that tries to improve the performance of Hadoop scheduling. Hadoop Fair Scheduler is proposed in [4] which divides resources using max-min fair sharing [5] to achieve fairness among users and jobs. The author proposed an algorithm in [6] that takes into account the interoperability of MapReduce tasks running on a node of the cluster, the algorithm tries to run a task from the list of pending tasks that is most compatible with the tasks already running on the TaskTracker. The algorithm computes an average task vector which represents task's average resource usage and whenever a TaskTracker has an empty slot for a task, a Task Selection algorithm checks if the task vector is compatible with the already running tasks on that TaskTracker. The algorithm proposed in [7] collects information about the running tasks and dynamically adjusts the slot share among users. In [8], the proposed algorithm allows the user to decide a set of TaskTrackers to execute a particular job and controls the number of tasks for a job that can run concurrently on a TaskTracker. An algorithm that dynamically reallocate slots between map and reduce slots is proposed in [9] as the need of the currently executed jobs, if there is insufficient map slots, the algorithm reallocate unused reduce slots to be map slots and vice versa. In [10], a framework called ARIA is proposed consists of a job profiler, performance model and SLO-based scheduler tries to meet the job completion time deadline associated with the job submitted. The authors in [11] propose a scheduler that uses estimated job arrival rates and mean job execution time to improve mean completion time. There are a scheduling algorithms that try to improve the Hadoop performance by enhancing data locality to avoid unnecessary data transmission as proposed in [4][12][13]. In [14] and [15], the authors proposed algorithms that improve the performance of Hadoop by improving the method of discovering the tasks that need to be speculatively executed.

All the above scheduling algorithms still run over a fixed total number of map/reduce slots for each TaskTracker over the run time neglecting the TaskTracker specifications and performance. This causes TaskTracker under or over utilization and poor performance. Reference [3] proposes an algorithm that uses job profiling information to dynamically adjust the number of slots on each machine, it introduces a 'job slot' which is a slot that is job specific and has an associated resource demand profile for map and reduce tasks, the proposed job scheduling technique relies on the use of job profiles containing information about the resource consumption for each job, to build a job profile the user should run a job in a sandbox environment with the same characteristics of the production environment, the job should run in isolation multiple times in the sandbox using different configurations. For example, if the bottleneck is CPU, the algorithm should choose a sum of 100% of CPU utilization from the jobs profiles, the algorithm can choose a four tasks each needs 25% CPU to run on the TaskTracker. We found that it's difficult in practical to run each job in isolation many times to produce the job profile before submitting to the Hadoop cluster. Another shortcoming of that algorithm is, an inaccurate profiles lead to resource under or over utilization thus the profile accuracy plays a role in the performance of the algorithm. So, we proposed our scheduler that considers resources heterogeneity and doesn't need any prior execution of each job as in [3]. We will describe our proposed scheduler in the following Section.

III. Proposed Algorithm

Our proposed scheduler considers resources heterogeneity and dynamically adapts to different nodes with different computing capabilities. Our scheduler dynamically extends/shrinks the capacity of each TaskTracker separately by controlling the number of map/reduce slots on each TaskTracker depending on the TaskTracker available resources, performance and the needs of the running jobs. The scheduler monitors the resources utilization over the run time for each node. If there is available/lack resources (CPU and memory) the scheduler will extend/shrink the capacity of the TaskTracker by increasing/decreasing the number of map/reduce slots of the TaskTracker. The user can choose the percentage of the resources utilization and the maximum/minimum number of slots on a TaskTracker via a scheduler configuration file. Our scheduler consists of two components described as follow:

- Resource Monitoring Procedure
- Slot Allocation Algorithm

The following is Table I which lists the notations used in the algorithm.

TABLE I. Algorithm Notations

Notation	Description
m_i/r_i	No. of map/reduce slots on TaskTracker i
U^*	CPU utilization threshold specified by the user
U_i	CPU utilization on TaskTracker i
mf_i/rf_i	No. of free map/reduce slots of TaskTracker i
$Tm_\bullet/Tr_\bullet$	No. of pending map/reduce tasks of all submitted jobs
m_{Max}/r_{Max}	Max no. of map/reduce slots on the TaskTracker configured by user
m_{Min}/r_{Min}	Min no. of map/reduce slots on the TaskTracker configured by user

The components of the scheduler will be described in details in the following subsections.

A. Resource Monitoring Procedure

This procedure runs on each TaskTracker to monitor TaskTracker resources utilization (CPU, memory and network bandwidth) over the run time. The procedure computes the average resources utilization during the heartbeat period, collects

processor information such as number of cores and frequency, and sends all these information to the JobTracker via the heartbeat message. These information will be used by the 2^{nd} component of the proposed scheduler which will be described in the following subsection.

B. Slot Allocation Algorithm

The Slot Allocation Algorithm is the main component of the proposed scheduler which runs on the JobTracker. The algorithm will be described by the following Pseudocode:

Algorithm 1 : Slot Allocation Algorithm

On a heartbeat reception from TaskTracker T_i
  if first heartbeat from T_i **then**
    Initialize $m_i \leftarrow$ no. of CPU cores of T_i
    Initialize $r_i \leftarrow 2$
  else
    if $U_i < U^*$ **then**
      if $mf_i = 0$ and $Tm_p > 0$ and $m_i < m_{Max}$ **then**
        $m_i \leftarrow m_i$ + 1
      else if $rf_i = 0$ and $Tr_p > 0$ and $r_i < r_{Max}$ **then**
        $r_i \leftarrow r_i$ + 1
      end if
    else
      if $mf_i > 0$ and $m_i > m_{Min}$ **then**
        $m_i \leftarrow m_i$ - 1
      else if $rf_i > 0$ and $r_i > r_{Min}$ **then**
        $r_i \leftarrow r_i$ - 1
      end if
    end if
  end if

The algorithm initializes the number of map slots of each TaskTracker to the number of cores/threads of the processor of the TaskTracker. So, TaskTracker with two cores processor will be initialized by two map slots, four cores processor will be initialized by four map slots, etc... In case of reduce slots the algorithm initializes the number of reduce slots by two slots on each TaskTracker. Over the run time the algorithm gets the average CPU and memory usage via the TaskTracker heartbeat and if the current workload needs map/reduce slots, the algorithm checks the available resources of the TaskTracker on heartbeat reception and compares the average usage with the threshold values configured by the user. If there are available resources on the TaskTracker, the algorithm increases the number of map/reduce slot (as the workload needs) of the TaskTracker by one more slot. To avoid TaskTracker overloading, if we found the average resource usage exceeds the threshold values and there is a free map/reduce slot(s) available on that TaskTracker, the algorithm decreases the number of map/reduce slots by one slot. The resources usage thresholds and maximum/minimum number of map/reduce slots per TaskTracker are configured by the user via a configuration file. The scheduler doesn't need any prior execution of jobs in isolation, we don't need job profiles submitted with the jobs. Our scheduler is adaptive and dynamic for homogenous and heterogenous resources. The evaluation of the proposed scheduler is presented in the following Section.

IV. Evaluation And Results

We implemented our scheduler on Hadoop by modifying the JobTracker, TaskTracker and TaskTrackerStatus classes to support collecting the required resources information. We modified the heartbeat class to allow sending these information to the JobTracker and heartbeatrespone classe to allow the JobTracker to send the assigned number of slots to the TaskTrackers. Also, we implemented the scheduler by extending the Task Scheduler interface.

A. Experimental Environment and Workload

We tested the proposed scheduler on a cluster consisting of four nodes (one master and three slaves) with different computing capabilities with specifications described in the following Table.

TABLE II. Experimental Environment

No. of nodes	CPU	Memory	Node(s) name
2	Intel Core i3 3.3 GHZ	4 GB	Master and Slave1
2	Intel Dual Core 2.5 GHZ	2 GB	Slave2 and Slave3

To evaluate our scheduler we used three representative applications used by some researchers to evaluate their scheduling algorithms, these applications are:

- WordCount: This application counts the occurrences of each work in the input file(s).
- PiEstimator: The 'PiEstimator' application uses quasi-Monte Carlo method to estimate the Pi value.
- TeraSort: The 'TeraSort' application is used to samples the input data and uses map/reduce to sort the data into a total order.

These applications are submitted with the sequence and data size described in Table III.

TABLE III. Jobs Specifications

Job Submission Sequence	Job Name	Data size/no. of samples
1	WordCount	runs on 10 GB of data
2	TeraSort	runs on 1 GB of data
3	PiEstimator	10000000 sample
4	TeraSort	runs on 1 GB of data

B. Results

We have tested our scheduler on single jobs and multiple jobs. On the single jobs experiments we have tested our scheduler *vs.* Hadoop Fair Scheduler configured with 4 map slots. Each time a different single job is submitted. We got the results shown in Fig. 1.

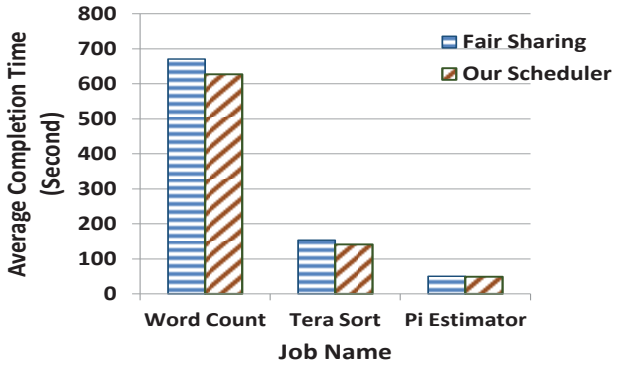

Fig. 1. Completion time of single jobs using the proposed scheduler and Hadoop Fair Scheduler

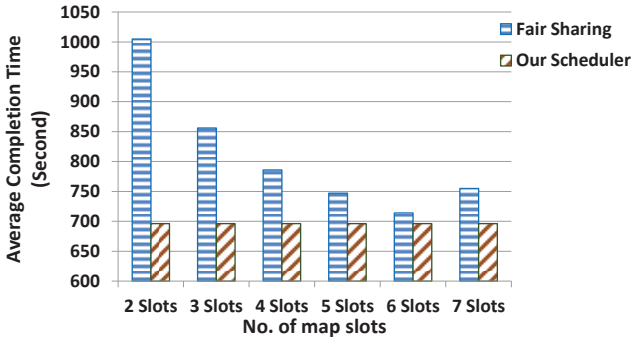

Fig. 2. The average completion time of our algorithm *vs.* Hadoop Fair Scheduler configured with different number of map slots

TABLE IV. COMPLETION TIME IMPROVEMENT PERCENTAGE

Job Name	Average Completion time (Second)		Improvement %
	Fair Sharing	Proposed Scheduler	
Word Count	671	627	6.6%
Tera Sort	147	141	4.1%
Pi Estimator	50	49	2%

From Fig.1, we can see that our scheduler got a less completion time than Hadoop Fair Scheduler. Table IV, extracted from Fig.1 shows the completion time and the percentage of improvement. We can find from Table IV that the difference in completion time is the least when running Pi Estimator job (just 1 second less in completion time), that's because the Pi Estimator job needs just 10 map slots and we already have 16 map slots in our cluster (4 nodes each configured with 4 map slots) when applying Hadoop Fair Scheduler. Approximately the same case when running the TeraSort job, it needs 16 map slots and we also have 16 map slots when applying Hadoop Fair Scheduler. The improvement of our scheduler increased in the case of WordCount job (44 second less in completion time). The Word Count job needs 170 map slots. We can conclude from this result that the effect of our scheduler increases in the case of high workload or bad/static configuration that may cause under/over utilization of resources. In the multi jobs experiments, we have two types of experiments. In both we show the completion time and the CPU utilization. In the First type, we compared our proposed scheduler with Hadoop Fair Scheduler each time configured with a different number of map slots for all TaskTrackers and the same number of jobs submitted. While in the second experiment, our scheduler was compared with the same algorithm each time with a different number of submitted jobs and the same number of slots.
In the first type of multiple jobs experiments, we ran the four jobs listed in Table III in sequence with approximately 5 seconds delay between each job and the previous job. We tested Hadoop Fair Scheduler configured each time with a different number of map slots on all TaskTrackers started by 2, each time increased by 1 until we reached to 7 map slots. The results are shown in Fig. 2.

From Fig. 2, we find that our scheduler achieved less completion time than the Hadoop Fair Scheduler configured with different number of map slots. We can see that when we increase the number of map slots using Hadoop Fair Scheduler, we get a better average completion time due to the extension of the node capacity and leveraging of the available resources. But after we configured the number of map slots to 7 slots on each node, the performance retracted and the average completion time increased, thats due to the overloading caused by running too much map tasks concurrently on the low performance nodes of our cluster. The reason of the good performance of our scheduler as shown in Table V, extracted from Fig. 2 that it monitors the performance of each node separately and finds the appropriate number of slots for that machine. The following figure illustrates the CPU utilization of the Master node when using Hadoop Fair Scheduler configured with 2 and 4 map slots, and our proposed scheduler.

TABLE V. COMPLETION TIME IMPROVEMENT PERCENTAGE

No. of Slots	Average Completion time (Second)		Improvement %
	Fair Sharing	Proposed Scheduler	
2	1005	696	30.7%
3	856	696	18.7%
4	786	696	11.5%
5	747	696	6.8%
6	714	696	2.5%
7	755	696	7.8%

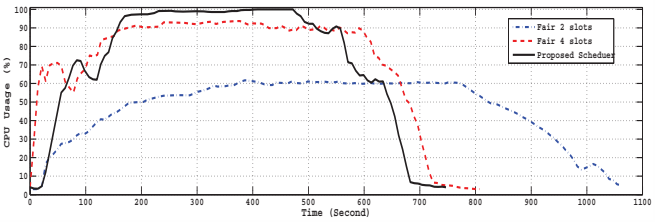

Fig. 3. CPU utilization using Hadoop Fair Scheduler configured with 2 and 4 map slots and our proposed scheduler

We can see from Fig. 3 that when we applied the Hadoop Fair Scheduler using 2 map slots we got about 60% CPU utilization and when using 4 map slots we got about 90 - 92 % CPU utilization. We find that our scheduler got better CPU utilization approximately 98 - 100 %.

In the second type of multiple jobs experiments, we configured the Hadoop Fair Scheduler to 2 map slots as it's the default number of slots configured in Hadoop and we changed the number of submitted jobs each time. We got the results shown in Fig. 4.

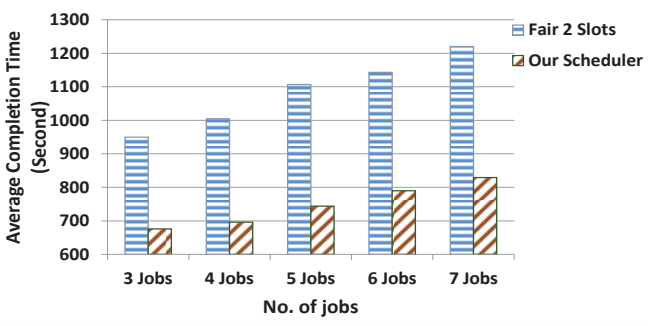

Fig. 4. The average completion time of our scheduler *vs.* Hadoop Fair Scheduler configured with 2 map slots when submitting different number of jobs

TABLE VI. COMPLETION TIME IMPROVEMENT PERCENTAGE

No. of Jobs	Average Completion time (Second)		Improvement %
	Fair Sharing	Proposed Scheduler	
3	950	676	28.8%
4	1005	696	30.7%
5	1107	744	32.8%
6	1143	790	30.9%
7	1220	829	32.0%

We can see from Fig. 4 that our scheduler performs in a less completion time than the Hadoop Fair Scheduler when submitting different number of jobs as Table VI, extracted from Fig. 4 illustrates the percentage of improvement.

Fig. 5 shows the CPU utilization of two different capabilities nodes (Master and Slave2) using Hadoop Fair Scheduler configured with 2 map slots for each node and our proposed scheduler. When using the Hadoop Fair Scheduler with a fixed number of slots (2 slots) as in Fig. 5 (a) we got unbalanced resources utilization on the both nodes. The node with the less H.W specifications (Slave2) gets more resources usage about 90 % CPU usage, while the node with the higher H.W specifications (Master) gets under utilized (about 50-64 % CPU usage). But when we applied our dynamic scheduler we got a balanced resources utilization as in Fig. 5 (b). Both of nodes didn't get over/under utilized. This show that our scheduler performs dynamically on all nodes depending on each node performance over the run time, which leads to high Hadoop performance and less completion time.

Fig. 6 illustrates the number of map slots over the run time on 2 nodes with different specifications, the Slave1 and the Slave2 with the specifications mentioned in Table II.

We can see from Fig. 6 that the number of map slots varies over run time depending on the node performance as we see that Slave1 node with 4 threads processor has been initialized by 4 map slots and the number of map slots increased over the run time to be 5, 6 and some times 7 map slots. Also, Slave2 node with 2 threads processor has been initialized by 2 map slots and the number of map slots increased and decreased over the run time depending on the node performance. Our scheduler adapts to the node performance to maintain the CPU usage to the level specified by the user, while avoiding node under/over utilization.

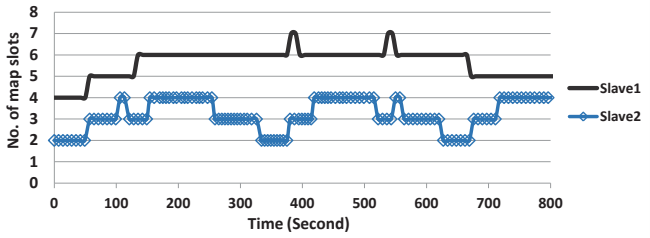

Fig. 6. No of map slots assigned by our scheduler on Slave1 and Slave2 nodes over the run time

V. CONCLUSION AND FUTURE WORK

In this paper we proposed a dynamic Hadoop scheduler that adapts to different nodes computing capabilities, performance over time and different jobs needs. The proposed scheduler extends/shrinks the capacity of each TaskTracker separately according to its performance over the run time instead of being fixed capacity as in the related work in Section II. We implemented our scheduler on Hadoop and tested the scheduler on a cluster consists of a set of nodes with different computing capabilities and different jobs. The results show that our scheduler has achieved higher resources utilization, less completion time and better avoiding of resources under/over utilization compared to the Hadoop Fair Scheduler. In Future work, we plan to extend our scheduler to support resources distribution among users/submitted jobs leveraging the knowledge of job completion time, jobs need and node performance to reach overall optimal performance which will lead to less completion time and balanced resources utilization. Also, we plan to test our proposed scheduler on a larger cluster on Amazon EC2 and study the performance of the scheduler with a larger number of submitted jobs.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce : Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "Apache hadoop." [Online]. Available: http://hadoop.apache.org/

[3] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguadé, "Resource-aware adaptive scheduling for mapreduce clusters," pp. 187–207, 2011.

[4] M. Zaharia and S. Shenker, "Delay scheduling : A simple technique for achieving locality and fairness in cluster scheduling," *Cluster Computing*, 2010.

[5] "Max-min fairness (wikipedia)." [Online]. Available: http://en.wikipedia.org/wiki/Max-min fairness

[6] R. Nanduri, N. Maheshwari, a. Reddyraja, and V. Varma, "Job aware scheduling algorithm for mapreduce framework," *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pp. 724–729, Nov. 2011.

[7] Y. Yao, J. Tai, B. Sheng, and N. Mi, "Scheduling heterogeneous mapreduce jobs for efficiency improvement in enterprise clusters," *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pp. 872–875, 2013.

[8] J. S. Manjaly and V. S. Chooralil, "Tasktracker aware scheduling for hadoop mapreduce," *2013 Third International Conference on Advances in Computing and Communications*, pp. 278–281, Aug. 2013.

[9] S. Tang, B.-S. Lee, and B. He, "Dynamic slot allocation technique for mapreduce clusters," *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pp. 1–8, 2013.

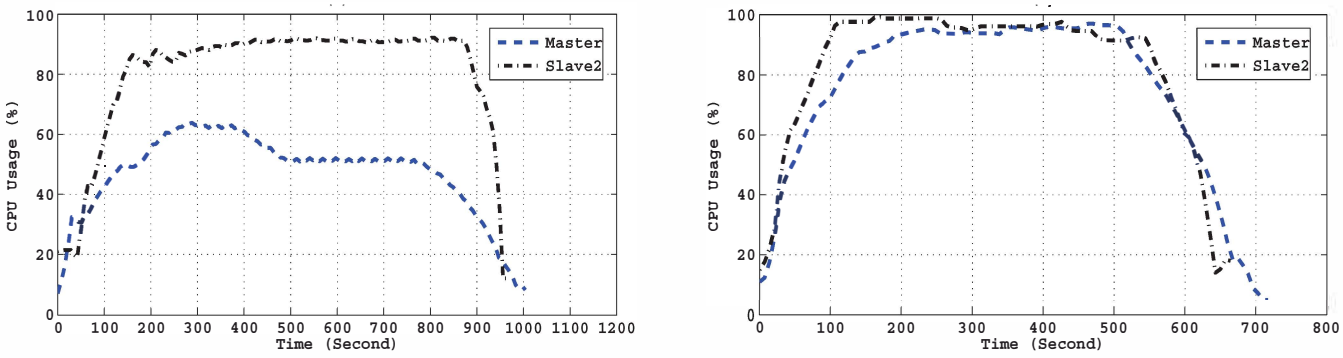

Fig. 5. CPU utilization of two different computing capabilities nodes using (a)Hadoop Fair Scheduler configured with 2 map slots and (b) Proposed algorithm

[10] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," *Proceedings of the 8th ACM international conference on Autonomic computing*, pp. 235–244, 2011.

[11] A. Rasooli and D. Down, "An adaptive scheduling algorithm for dynamic heterogeneous hadoop systems," *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp. 30–44, 2011.

[12] C. He, Y. Lu, and D. Swanson, "Matchmaking: A new mapreduce scheduling technique," *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 40–47, 2011.

[13] J. Tan, S. Meng, X. Meng, and L. Zhang, "Improving reducetask data locality for sequential mapreduce jobs," *INFOCOM, 2013 Proceedings IEEE*, pp. 1627–1635, 2013.

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." *OSDI*, vol. 8, no. 4, p. 7, 2008.

[15] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, "Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment," *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 2736–2743, 2010.