2011

Matchmaking: A New MapReduce Scheduling Technique

Chen He
University of Nebraska-Lincoln, che@cse.unl.edu

Ying Lu
University of Nebraska-Lincoln, ying@unl.edu

David Swanson
University of Nebraska-Lincoln, dswanson@cse.unl.edu

Matchmaking: A New MapReduce Scheduling Technique

Chen He Ying Lu David Swanson
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, U.S.
{che,ylu,dswanson}@cse.unl.edu

***Abstract*— MapReduce is a powerful platform for large-scale data processing. To achieve good performance, a MapReduce scheduler must avoid unnecessary data transmission by enhancing the data locality (placing tasks on nodes that contain their input data). This paper develops a new MapReduce scheduling technique to enhance map task's data locality. We have integrated this technique into Hadoop default FIFO scheduler and Hadoop fair scheduler. To evaluate our technique, we compare not only MapReduce scheduling algorithms with and without our technique but also with an existing data locality enhancement technique (i.e., the delay algorithm developed by Facebook). Experimental results show that our technique often leads to the highest data locality rate and the lowest response time for map tasks. Furthermore, unlike the delay algorithm, it does not require an intricate parameter tuning process.**

Keywords—MapReduce; Hadoop; data locality; scheduling technique

I. INTRODUCTION

MapReduce is a framework used by Google for processing huge amounts of data in a distributed environment [1] and Hadoop [2] is Apache's open source implementation of the MapReduce framework. Due to the simplicity of the programming model and the run-time tolerance for node failures, MapReduce is widely used for not only commercial applications but also scientific computations. Facebook uses a Hadoop cluster composed of hundreds of nodes to process terabytes of user data. The New York Times rents a Hadoop cluster from Amazon EC2 [3] to convert millions of articles. Michael C. Schatz [4] introduced MapReduce to parallelize *blast* which is a DNA sequence alignment program and achieved 250 times speedup. As MapReduce clusters get popular, their scheduling becomes increasingly important. In a MapReduce cluster, data are distributed to individual nodes and stored in their disks. To execute a map task on a node, we need to first have its input data available on that node. Since transferring data from one node to another takes time and delays task execution, an efficient MapReduce scheduler must avoid unnecessary data transmission.

In this paper, we focus on the problem of decreasing data transmission in a MapReduce cluster and we develop a scheduling technique to improve map tasks' data locality rate. For a given execution of MapReduce workload, the data locality rate is defined in this paper as the ratio between the numbers of local map tasks and all map tasks, where a local map task refers to a task that has been executed on a node that contains its input data. A low data locality rate means more data transfer between machines and higher network traffic. To avoid unnecessary data transfer, our scheduling technique aims to achieve high data locality rate and also short response time for MapReduce clusters.

Existing MapReduce algorithms provide some mechanisms to improve the data locality. For instance, to assign map tasks to a node, the Hadoop default FIFO (first-in-first-out) scheduler always picks the first job in the waiting queue and schedules its local map tasks (i.e., tasks with input data stored in the node). If the job does not have any map task local to the node, only one of its non-local map tasks will be assigned to the node at a time. However, due to FIFO scheduler's inherent deficiencies (like following the strict FIFO job order for assigning tasks), this mechanism can only slightly improve the data locality.

Zaharia et al. [5] have developed a delay technique to improve the data locality rate. With this technique, a MapReduce scheduler breaks the strict job order when assigning map tasks to a node. That is, if the first job does not have a local map task, the scheduler can delay it and assign another job's local map tasks. A maximum delay time D is specified. Only when a job has been delayed for more than D time units will the scheduler assign the job's non-local map tasks. For the delay algorithm, the maximum delay time D is a critical factor. It is configurable but may need to vary for different workloads and hardware environments.

This paper develops a new technique to enhance the data locality. The main idea of the technique is as follows. To assign tasks to a node, local map tasks are always preferred over non-local map tasks, no matter which job a task belongs to, and a locality marker is used to mark nodes and to ensure each node a fair chance to grab its local tasks. Experiments are carried out to evaluate the aforementioned techniques and experimental results show that our technique leads to the highest data locality rate and the lowest response time for map tasks. Unlike the delay algorithm, our technique does not require the tuning of the delay parameter.

978-0-7695-4622-3 2011

DOI 10.1109/CloudCom.2011.16

The remainder of this paper is organized as follows. Section 2 presents the background. In Section 3, we describe our scheduling technique, which is evaluated in Section 4. Section 5 presents the related work and Section 6 concludes this paper.

II. BACKGROUND

Hadoop [2] is a widely-used open source implementation of Google MapReduce [1]. In this section, we briefly describe how a Hadoop cluster works since other MapReduce-style clusters work similarly. In later parts of this paper, we will thus use the terms "Hadoop cluster" and "MapReduce cluster" interchangeably. A MapReduce cluster is often composed of many commodity PCs, where one PC acts as the master node and others as slave nodes. A Hadoop cluster uses Hadoop Distributed File System (HDFS) [6] to manage its data. It divides each file into small fixed-size (e.g., 64 MB) blocks and stores several (e.g., 3) copies of each block in local disks of cluster machines. A MapReduce [1] computation is comprised of two stages, map and reduce, which take a set of input key/value pairs and produce a set of output key/value pairs. When a MapReduce job is submitted to the cluster, it is divided into M map tasks and R reduce tasks, where each map task will process one block (e.g., 64 MB) of input data.

A Hadoop cluster uses slave nodes to execute map and reduce tasks. There are limitations on the number of map and reduce tasks that a slave node can accept and execute simultaneously. That is, each slave node has a fixed number of map slots and reduce slots. Periodically, a slave node sends a heartbeat signal to the master node. Upon receiving a heartbeat from a slave node that has empty map/reduce slots, the master node invokes the MapReduce scheduler to assign tasks to the slave node. A slave node who is assigned a map task reads the contents of the corresponding input data block, parses input key/value pairs out of the block, and passes each pair to the user-defined map function. The map function generates intermediate key/value pairs, which are buffered in memory, and periodically written to the local disk and partitioned into R regions by the portioning function. The locations of these intermediate data are passed back to the master node, which is responsible for forwarding these locations to reduce tasks. A reduce task uses remote procedure calls to read the intermediate data generated by the M map tasks of the job. Each reduce task is responsible for a region (partition) of intermediate data. Thus, it has to retrieve its partition of data from all slave nodes that have executed the M map tasks. This process is called shuffle, which involves many-to-many communications among slave nodes. The reduce task then reads in the intermediate data and invokes the reduce function to produce the final output data (i.e., output key/value pairs) for its reduce partition [1].

Since network bandwidth is a relatively scarce resource in a MapReduce cluster, we can conserve it by taking advantage of the fact that the input data is stored in the local disks of machines that make up the cluster [1]. Thus, a MapReduce scheduler often takes input files' location information into account and attempts to schedule a map task on a slave node that contains a replica of the corresponding input data block. This way, map tasks' data locality rate can be improved, where most input data is read locally and consumes no network bandwidth.

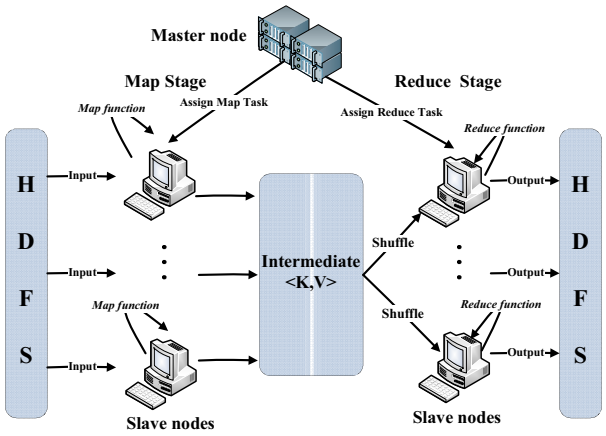

Figure 1. Hadoop Framework

A. Hadoop default FIFO scheduler

The Hadoop default FIFO scheduler has already taken data locality into account. When a slave node with empty map slots sends the heartbeat signal, the MapReduce scheduler checks the first job in the queue. If the job has map tasks whose input data blocks are stored in the slave node, the scheduler assigns the node one of these local tasks. If a slave node has more unused map slots, the scheduler will keep assigning local tasks to the node. However, if the scheduler can no longer find a local task from the first job, it assigns the node one and only one non-local task during this heartbeat interval, no matter how many free slots the node has.

This default FIFO scheduler, however, has deficiencies. First of all, it follows the strict FIFO job order to assign tasks, which means it will not allocate any task from other jobs if the first job in the queue still has an unassigned map task. This scheduling rule has a negative effect on the data locality because another job's local tasks cannot be assigned to the slave node unless the first job has all its map tasks (many of which are non-local to the node) scheduled.

Secondly, the data locality is randomly decided by the heartbeat sequence of slave nodes. If we have a large cluster that executes many small jobs, the data locality rate could be quite low. As mentioned, in a MapReduce cluster, tasks are assigned to a slave node in response to the node's heartbeat. With the FIFO scheduler, heartbeats are also processed in a FIFO order and a node is assigned a non-local map task when there is no local task from the first job. In a large cluster many nodes heartbeat simultaneously. However, a

small job has less input data that are stored in a small number of nodes. It is thus a high probability event that the scheduler assigns tasks to slave nodes that do not have the small job's input data but give heartbeats first. For example, if we execute a job of 5 map tasks on a MapReduce cluster of 100 slave nodes, it is unlikely to get a high locality rate. Since each map task needs one input data block, which by default has 3 replicas stored in 3 nodes, at most 15 out of 100 nodes have input data for the job, i.e., the job's tasks are all non-local to at least 85 nodes. A slave node with empty map slots that sends in a heartbeat first will always be assigned at least one map task, local or non-local. It is highly likely that the job's tasks will be assigned to many of those 85 nodes which do not have the input data blocks before a node even gets a chance to grab a local task from the job.

B. Delay scheduling

Zaharia et al. [5][7] have developed a delay scheduling algorithm to improve the data locality rate of Hadoop clusters. It relaxes the strict job order for task assignment and delays a job's execution if the job has no map task local to the current slave node. To assign tasks to a slave node, the delay algorithm starts the search at the first job in the queue for a local task. If not successful, the scheduler delays the job's execution and searches for a local task from succeeding jobs. A maximum delay time *D* is set. If a job has been skipped long enough, i.e., longer than *D* time units, its non-local tasks will then be assigned for execution. With the delay scheduling algorithm, a job's execution is postponed to wait for a slave node that contains the job's input data. Here, the delay time *D* is a key parameter. By default, it is set at 1.5 times the slave node's heartbeat interval. However, to obtain the best performance for the delay scheduling algorithm, we have to choose an appropriate *D* value. If the value is set too large, job starvations may occur and affect performance. On the contrary, a too small *D* value allows non-local tasks to be assigned too fast. For different kinds of workloads and hardware environments, the best delay time may vary. To get an optimal delay time always needs careful tuning.

In addition, this delay algorithm allows a node to obtain multiple non-local map tasks in a heartbeat interval if the node has more than one free slot. In some situations, this algorithm could perform worse than the FIFO scheduler's locality enhancement policy because the latter only allows one non-local task to be assigned to a node in a heartbeat interval.

Although first developed to improve the data locality of the Hadoop fair scheduler [14], delay scheduling is applicable beyond fair sharing, in general, applicable to any scheduling policy (e.g., FIFO) that defines an order in which jobs should be given resources [5].

III. MATCHMAKING SCHEDULING ALGORITHM

This section presents our new technique for enhancing the data locality in MapReduce clusters. The main idea behind our technique is to give every slave node a fair chance to grab local tasks before any non-local tasks are assigned to any slave node. Since our algorithm tries to find a match, i.e., a slave node that contains the input data, for every unassigned map task, we call our new technique the matchmaking scheduling algorithm.

First of all, like the delay scheduling algorithm, our matchmaking algorithm also relaxes the strict job order for task assignment. If a local map task cannot be found in the first job, the scheduler will continue searching the succeeding jobs. Second, in order to give every slave node a fair chance to grab its local tasks, when a node fails to find a local task in the queue for the first time in a row, no non-local task will be assigned to the node. That is, the node gets no map task for this heartbeat interval. Since during a heartbeat interval, all slave nodes with free map slots have likely given their heartbeats and been considered for local task assignment, when a node fails to find a local task for the second time in a row (i.e., still no local task a heartbeat interval later), to avoid wasting computing resources, the matchmaking algorithm will assign the node a non-local task. This way, our algorithm achieves not only high data locality rate but also high cluster utilization. To enforce the aforementioned rule, our algorithm gives every slave node a locality marker to mark its status. If none of the jobs in the queue has a map task local to a slave node, depending on this node's marked value, the matchmaking algorithm will decide whether or not to assign the node a non-local task. Third, our matchmaking algorithm allows a slave node to take at most one non-local task every heartbeat interval. At last, all slave nodes' locality markers will be cleared when a new job is added to the job queue. Because a new job may comprise new local tasks for some slave nodes, upon the new job's arrival, our algorithm resets the status of all nodes and again starts the all-to-all task-to-node matchmaking process. Tables 1 and 2 give the pseudo code of our algorithm. Like delay scheduling algorithm, our matchmaking algorithm is applicable to any scheduling policy (e.g., FIFO or fair sharing scheduling) that defines an order in which jobs should be given resources.

IV. EVALUATION

To evaluate our matchmaking scheduling algorithm, we compare it with the Hadoop default FIFO scheduler and the delay scheduling algorithm. Two metrics, i.e., map tasks' *data locality rate* and *average response time*, are used for evaluation.

We run experiments in a private cluster of 1 head node and 30 slave nodes that are configured as one rack. We modify Hadoop-0.21 and integrate our matchmaking algorithm with both Hadoop FIFO scheduler and Hadoop fair scheduler. The cluster is configured with a block size of 128MB, which follows Facebook's Hadoop cluster block

size configuration [5]. Table 3 lists our Hadoop cluster hardware environment and configuration.

TABLE 1. SCHEDULING ALGORITHM

Algorithm 1: Matchmaking Scheduling Algorithm

```
for each node i of the N slave nodes do
   set LocalityMarker[i]=null
end for

Upon receiving a heartbat from node i:
while node i has free slots, i.e., its free slot count s>0

   set previousMarker=LocalityMarker[i]

   for each job j in the JobQueue do
      if job j has an unassigned local task t then
         assign t to node i
         set s=s-1
         if LocalityMarker[i]==null then
               LocalityMarker[i]=1
         else LocalityMarker[i]+=1
         end if
         break for
      else continue
      end if
   end for

   if previousMarker==LocalityMarker[i] then
      set LocalityMarker[i]=0        //mark this node
      break while
   else if LocalityMarker[i]==0 then
      assign node i a non-local task t' from the first job in
the JobQueue
      set s=s-1
      break while
   end if
end while
```

TABLE 2. LOCALITY MARKER CLEANING ALGORITHM

Algorithm 2: Locality Marker Cleaning Algorithm

```
When a new job j is added into the JobQueue:
for each node i of the N slave nodes do
   set LocalityMarker[i]=null
end for
```

A. Experimental Environment

To evaluate our matchmaking algorithm, we create a submission schedule that is similar to the one used by Zaharia et al. [5]. Zaharia et al. [5] generated a submission schedule for 100 jobs by sampling job inter-arrival times and input sizes from the distribution seen at Facebook over a week in October 2009. By sampling job inter-arrival times at random from the Facebook trace, they found that the distribution of inter-arrival times was roughly exponential with a mean of 14 seconds.

They also generated job input sizes based on the Facebook workload, by looking at the distribution of number of map tasks per job at Facebook and creating datasets with the correct sizes (because there is one map task per 128 MB input block). Job sizes were quantized into nine bins, listed in Table 4 [5], to make it possible to compare jobs in the same bin within and across experiments. Our submission schedule has similar job sizes and job inter-arrival times. In particular, our job size distribution follows the first six bins of job sizes shown in Table 4, which cover about 89% of the jobs at the Facebook production cluster. Because most jobs at Facebook are small and our test cluster is limited in size, we exclude those jobs with more than 300 map tasks. Like the schedule in [5], the distribution of inter-arrival times is exponential with a mean of 14 seconds, making our submission schedule totally 21 minutes long.

TABLE 3. EVALUATION ENVIRONMENT

Nodes	Quantity	Hardware and Hadoop Configuration
Master node	1	2 single-core 2.2GHz Optron-64 CPUs, 8GB RAM, 1Gbps Ethernet
Slave nodes	30	2 single-core 2.2GHz Optron-64 CPUs, 4GB RAM, 1 Gbps Ethernet, 1 rack, 2 map and 1 reduce slots per node

We generate 100 input data blocks in Hadoop Distributed File System (HDFS). The popularity of blocks is assumed to follow a uniform distribution. That is, when a job requests a block, it is evenly likely to be any one of the blocks stored in HDFS. Each of the blocks has 2 replicas. We distribute and store these 200 block replicas evenly in 30 slave nodes, ensuring no two replicas of a block be stored in the same node. As a result, every slave node contains about 6 (or 7) blocks. By uniformly distributing blocks among our cluster nodes, we avoid hotspots of data requests.

We use our submission schedule for two application workloads. One is *loadgen* which is a test example from the Hadoop test package. It loads input data and outputs a fraction of the data intact. This application has been used as a test workload for the delay algorithm [5][7]. The other application we adopt is *wordcount* which is a classic example of Hadoop applications.

As mentioned, we have modified Hadoop-0.21 and integrated our matchmaking algorithm with both Hadoop FIFO scheduler and Hadoop fair scheduler.

In our experiments, we always configure the cluster to have just one job queue. With Hadoop fair scheduler, all jobs in a queue are scheduled following either fair sharing or FIFO scheduling rule. With fair sharing scheduling, resources are assigned to jobs such that all jobs get, on average, an equal share of resources over time. We have tested the performance of delay algorithm within Hadoop fair scheduler. Depending on the applied scheduling rule (FIFO or fair sharing), we have two different versions: FIFO with delay algorithm and Fair with delay algorithm. Since we have tested our matchmaking algorithm within Hadoop FIFO scheduler, when testing matchmaking algorithm within Hadoop fair scheduler, only the fair sharing scheduling rule is applied.

We thus run each workload under five schedulers: Hadoop FIFO scheduler, Hadoop FIFO scheduler with matchmaking algorithm, FIFO with delay algorithm, Fair with delay algorithm, and Fair with matchmaking algorithm.

TABLE 4. DISTRIBUTION OF JOB SIZES (IN TERMS OF NUMBER OF MAP TASKS) AT FACEBOOK [5]

Bin	#Maps	%Jobs at Facebook	#Maps in Benchmark	# of jobs in Benchmark
1	1	39%	1	38
2	2	16%	2	16
3	3-20	14%	10	14
4	21-60	9%	50	8
5	61-150	6%	100	6
6	151-300	6%	200	6
7	301-500	4%	400	4
8	501-1500	4%	800	4
9	>1501	3%	4800	4

For the delay algorithm, we need to configure the maximum delay time *D*. In our experiments, a total of 8 different *D* values are chosen. They are from 0.1 to 10 times the slave node's heartbeat interval. Since we configure the heartbeat interval to be 3 seconds long, the maximum delay time *D* changes from 0.3 to 30 seconds.

To eliminate the possible randomness of cluster hardware status, every point shown in the figures is the average of three runs.

B. Experiments

We first use the data locality rate to measure the performance of the following three schedulers: Hadoop FIFO scheduler, Hadoop FIFO scheduler with matchmaking algorithm, and FIFO with delay algorithm. Given a workload execution, the data locality rate is defined as,

$$\text{Data Locality Rate} = \frac{l}{n} \quad (1)$$

where l is the number of local map tasks and n is the total number of map tasks. Our experimental results on data locality rate with the two application workloads are shown in Figures 2 and 3. As we can see, the data locality rate achieved with the delay algorithm increases with the maximum delay time *D*. The longer a job is allowed to be delayed, the higher the probability that the job finds slave nodes that contain the input data blocks.

Figures 2 and 3 also show that the FIFO scheduler leads to the worst performance, i.e., the lowest data locality rate. However, when we integrate our matchmaking technique with the FIFO scheduler, the algorithm achieves the highest data locality rate, better than any of those achieved with the delay algorithm of different *D* values.

To evaluate the algorithms' performance only via the data locality rate is not enough since we can easily design an algorithm that enforces the constraint that all tasks have to be executed on slave nodes that contain their input data, leading to 100% data locality rate but also long response time for map tasks due to the long delay required to satisfy the strict constraint. Therefore, we also evaluate our algorithms by another metric: the average response time of all map tasks. Figures 4 and 5 present the experimental results. As shown in the figures, when we run the workloads with the FIFO scheduler, we get the longest average response time for map tasks. After enhancing the FIFO scheduler with our matchmaking algorithm, we reduce the average response time significantly.

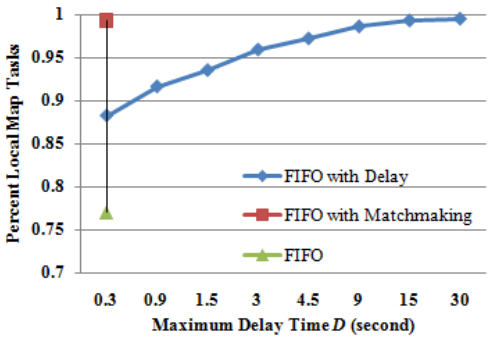

Figure 2. *Loadgen* Workload: Data Locality Rate

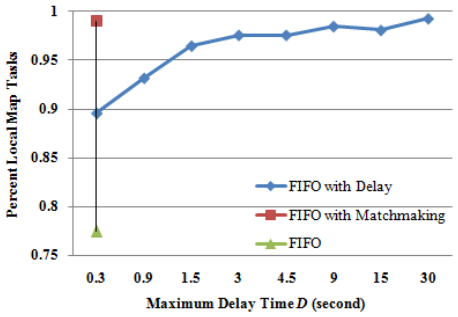

Figure 3. *WordCount* Workload: Data Locality Rate

For the delay algorithm, although the higher the *D* value, the better the data locality rate (see Figures 2 and 3), the relationship between the *D* value and the average response time is not so straightforward. When running the *loadgen* workload, the average response time varies with the *D* value, e.g., getting smaller when *D* increases from 0.3 to 1.5 seconds but longer when D increases from 1.5 to 3 seconds (see Figure 4). The lowest average response time is achieved when the maximum delay time is set at 30 seconds (see Figures 4 & 7-*loadgen*). But, that is not the optimal *D* value when running the *wordcount* workload. As shown in Figure 5 (and also in Figure 7-*wodcount*), when *D* = 9 seconds, we get the best average response time for the w*ordcount* workload. In neither cases, the default configuration (i.e., D = 4.5 seconds, 1.5 times the heartbeat interval) leads to the best performance. This group of experiments demonstrate that for different workloads, the best delay parameter varies, indicating the necessity of parameter tuning for the delay

algorithm. However, our matchmaking algorithm does not require this intricate parameter tuning process. For both workloads, the FIFO scheduler with our matchmaking algorithm achieves the lowest average response time, better than that achieved by the optimally-configured delay algorithm.

Let t_{avg} represent the average response time of all map tasks. It equals to the summation of two parts. That is,

$$t_{avg} = R_l t^l_{avg} + (1 - R_l) t^{nl}_{avg} \quad (2)$$

where R_l denotes the data locality rate, t^l_{avg} represents the average response time of all local map tasks, and t^{nl}_{avg} the average response time of all non-local map tasks.

Because network bandwidth is a relatively scarce resource in a MapReduce cluster [1] and the network data transferring rate is slower than the disk access rate, a local map task's execution is often much faster than that of a non-local map task. Therefore, according to Equation (2), increasing the data locality rate R_l tends to decrease the average response time of all map tasks t_{avg}. On the other hand, with the delay algorithm, as the maximum delay time D increases, a job and its tasks' execution is allowed to be delayed for a longer time. As a result, although R_l increases, both t^l_{avg} and t^{nl}_{avg} increase as well, leading to the potential increase of t_{avg}. This explains why map tasks' average response time does not decrease monotonically with the increase of the maximum delay time D.

So far, we have used experiments to compare three schedulers: Hadoop FIFO scheduler, Hadoop FIFO scheduler with matchmaking algorithm, and FIFO with delay algorithm. The results show that the FIFO scheduler with matchmaking algorithm achieves the highest locality rate and the lowest map task response time without the parameter tuning hassle. Next, to further compare the delay algorithm and our matchmaking algorithm, we integrate the matchmaking algorithm into Hadoop fair scheduler and compare the following two schedulers: Fair with delay algorithm and Fair with matchmaking algorithm.

Figures 6 and 7 show the data locality rate and the map tasks' average response time for the Hadoop fair schedulers.

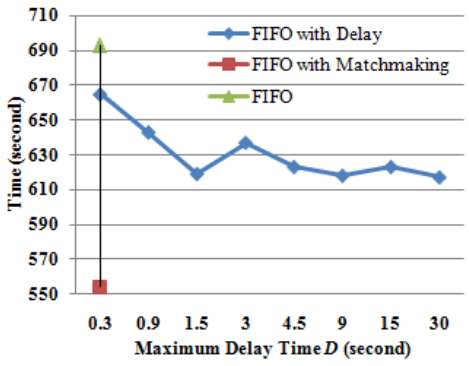

Figure 4. *Loadgen* Workload: Map Tasks' Average Response Time

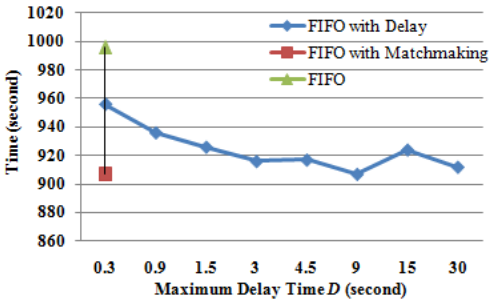

Figure 5. *WordCount* Workload: Map Tasks' Average Response Time

We can see that when integrated with the fair sharing scheduling, our matchmaking algorithm still achieves better data locality rates and near-optimal average response times. More importantly, our algorithm achieves this great performance without the necessity of parameter tuning.

V. Related Work

Due to the increasing importance of MapReduce clusters, recently there have been multiple studies on MapReduce scheduling.

MapReduce clusters can deal with node failures automatically. If a node fails to give a heartbeat within a timeout period, a MapReduce cluster will re-schedule the node's tasks to different nodes. Similarly, if a task's execution progresses slowly, a MapReduce cluster will run a speculative copy of this task on another node. This mechanism is called speculative execution. It prevents a job from being delayed by the worst performing node. Google has announced that this mechanism can improve a job's response time by 44% [1]. However, Hadoop's scheduler implicitly assumes that cluster nodes are homogeneous and tasks make progress linearly, and uses these assumptions to decide when to speculatively re-execute tasks that appear to be stragglers [9]. To overcome this limitation and make the speculative execution mechanism effective in heterogeneous environments, researchers then developed LATE (Longest Approximate Time to End) scheduler [9] and SAMR (Self-Adaptive MapReduce Scheduling) algorithm [10].

Yahoo! developed a multi-queue scheduler called Capacity Scheduler [11] for Hadoop clusters, where every queue is guaranteed a fraction of the capacity. Within a queue, it supports job priorities but no job pre-emption is allowed. To prevent one or more users from occupying all resources of a queue, each queue enforces a limit on the percentage of resources allocated to a user at any given time, if there is competition for resources.

The fair scheduler [14] also supports multiple queues (also called pools). Jobs are organized into pools and resources are fairly divided between these pools. By default, there is a separate pool for each user, so that each user gets an equal share of the cluster. Within each pool, jobs can be scheduled using either fair sharing or FIFO scheduling. Fair sharing scheduling is a method of assigning resources to

jobs such that all jobs get, on average, an equal share of resources over time. When there is a single job running, that job uses the entire cluster. When other jobs are submitted, task slots that free up are assigned to the new jobs, so that each job gets roughly the same amount of CPU time. Unlike the default Hadoop FIFO scheduler, which forms a queue of jobs based on job arrival times, this lets short jobs finish in reasonable time while not starving long jobs. It is also an easy way to share a cluster between multiple users [14].

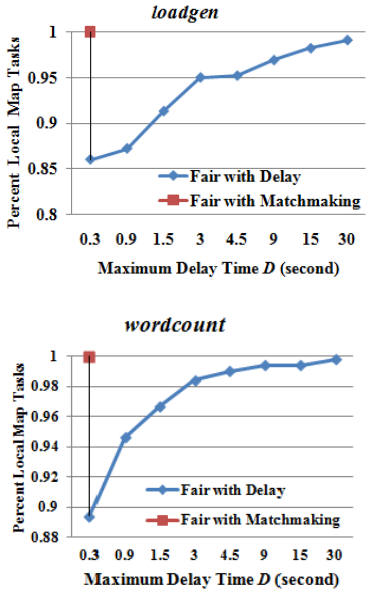

Figure 6. Fair Scheduler: Data Locality Rate

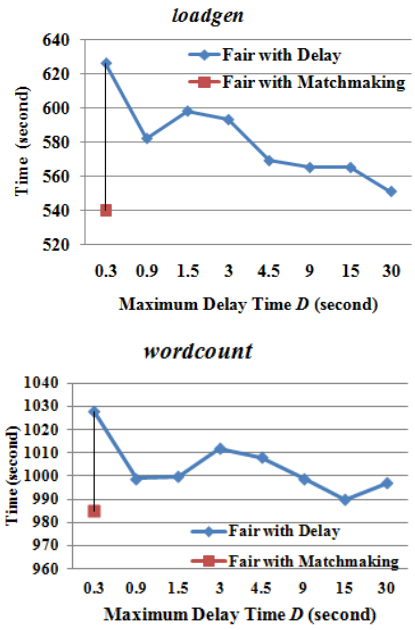

Figure 7. Fair Scheduler: Map Tasks' Average Response Time

To improve MapReduce clusters' data locality, researchers have used technologies like prefetching [15] or node status prediction [8]. The one that is most closely related to our work is the delay scheduling algorithm [5], which was first developed to improve the data locality of Hadoop fair scheduler [14].

Some MapReduce applications have deadlines. J. Polo et al. [12] developed a scheduler that focuses on MapReduce jobs that have soft deadlines. It estimates jobs' execution times and tries to let jobs satisfy their deadlines by scheduling resources according to the estimated finishing times. Kamal Kc et al. [13] created a scheduler that works for MapRedeuce applications with hard deadlines. It also estimates the job finishing time according to current resources in a MapReduce cluster. The difference is if a job cannot finish before the hard deadline, the scheduler will not execute the job and will instead inform the user to adjust the job deadline.

VI. Conclusion

In this paper, we develop a new matchmaking algorithm to improve the data locality rate and the average response time of MapReduce clusters. We have carried out experiments to compare not only MapReduce scheduling algorithms with and without our matchmaking algorithm but also with an existing data locality enhancement technique (i.e., the delay algorithm [5]). Experimental results demonstrate that our matchmaking algorithm can often obtain the highest data locality rate and the lowest average response time for map tasks. Furthermore, our matchmaking algorithm does not need any parameter tuning.

Acknowledgements

The authors acknowledge support from NSF award 1018467. This work was completed utilizing the Holland Computing Center of the University of Nebraska.

Reference

[1] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". Commun. ACM, 51(1):107–113, 2008.
[2] Apache Hadoop. http://hadoop.apache.org.
[3] Amazon EC2. http://aws.amazon.com/ec2/
[4] M.C. Schatz, "BlastReduce: high performance short read mapping with MapReduce". http://www.cbcb.umd.edu/software/blastreduce/.
[5] M. Zaharia et al. "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling". In EuroSys, 2010.
[6] HDFS. http://hadoop.apache.org/hdfs/
[7] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," EECS Department, University of California, Berkeley, Tech. Rep., Apr 2009.
[8] X. Zhang, Z. Zhong, S. Feng, B. Tu, J. Fan, "Improving Data Locality of MapReduce by Scheduling in Homogeneous Computing Environments", in 9th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 120-126, 2011.
[9] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, "Improving MapReduce performance in heterogeneous

environments", in: Proc. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, San Diego,USA, Dec. 2008.

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

[11] Capacity Scheduler http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html

[12] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguade and, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in Network Operations and Management Symposium (NOMS), 2010 IEEE, 2010, pp. 373 –380.

[13] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 388 –392, 2010.

[14] Fair Scheduler, http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html

[15] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng. "HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment". In Proc. CLUSTER'10, pp. 1–8, 2009.