ESAMR: An Enhanced Self-Adaptive MapReduce Scheduling Algorithm

Xiaoyu Sun, Chen He and Ying Lu
Department of Computer Science and Engineering,
University of Nebraska-Lincoln, Lincoln, NE 68588-0115, U.S.A.
Email: {xsun, che, ylu}@cse.unl.edu

***Abstract*—MapReduce is a programming model and an associated implementation for processing and generating large data sets. Hadoop is an open-source implementation of MapReduce, enjoying wide adoption, and is used not only for batch jobs but also for short jobs where low response time is critical. However, Hadoop's performance is currently limited by its default task scheduler, which 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. In practice, the homogeneity assumption does not always hold. Longest Approximate Time to End (LATE) is a MapReduce scheduling algorithm that takes heterogeneous environments into consideration. It, however, adopts a static method to compute the progress of tasks. As a result neither Hadoop default nor LATE schedulers perform well in a heterogeneous environment. Self-adaptive MapReduce Scheduling Algorithm (SAMR) uses historical information to adjust stage weights of map and reduce tasks when estimating task execution times. However, SAMR does not consider the fact that for different types of jobs their map and reduce stage weights may be different. Even for the same type of jobs, different datasets may lead to different weights. To this end, we propose ESAMR: an Enhanced Self-Adaptive MapReduce scheduling algorithm to improve the speculative re-execution of slow tasks in MapReduce. In ESAMR, in order to identify slow tasks accurately, we differentiate historical stage weights information on each node and divide them into k clusters using a k-means clustering algorithm and when executing a job's tasks on a node, ESAMR classifies the tasks into one of the clusters and uses the cluster's weights to estimate the execution time of the job's tasks on the node. Experimental results show that among the aforementioned algorithms, ESAMR leads to the smallest error in task execution time estimation and identifies slow tasks most accurately.**

Keywords: MapReduce, speculative task re-execution, heterogeneity

I. Introduction

In Today's world, data is growing exponentially, doubling its size every three years [1]. Huge amounts of data are being generated from digital media, web authoring, scientific instruments, physical simulations, and so on. Effectively storing, querying, analyzing, understanding, and utilizing these huge data sets presents one of the grand challenges to the computing industry and research community. A popular solution [2], [3], [4] is to build data-center scale computer systems to meet the high storage and processing demands of these applications. Such a system is composed of hundreds or thousands of commodity computers connected through a local area network housed in a data center. It has a much larger scale than a traditional computer cluster, while enjoying better and more predictable network connectivity than wide area distributed computing systems.

One of the most popular programming paradigms on data-center scale computer systems is the MapReduce programming model [2]. MapReduce is a programming model and an associated implementation for processing and generating large data sets [2]. It was first developed at Google by Jeffrey Dean and Sanjay Ghemawat. Under this model, an application is implemented as a sequence of MapReduce operations, each consisting of a map stage and a reduce stage that process a large number of independent data items. The system supports automatic parallelization, distribution of computations, task execution, and fault tolerance in hopes that application developers can focus on the design and implementation of applications without worrying about these complex system issues. Being a simple programming model, MapReduce has achieved great successes in various applications [5] [6] [7] [8] [9] [10] [11] [12] [13]. Hadoop[14], created by Doug Cutting [15], is an open source implementation of the MapReduce framework. It makes MapReduce framework widely accessible.

As mentioned, a key benefit of MapReduce is its automatic handling of failures, which hides the complexity of fault-tolerance [16] [14] [17] from application developers. If a node crashes, MapReduce re-executes failed tasks on a different machine. Equally importantly, when an available node performs poorly, a condition referred to as a straggler, MapReduce speculatively re-executes a straggler task on another machine to finish the computation faster. Without this speculative execution mechanism [18], a job would be as slow as the misbehaving task.

In this work, we address the problem of how to robustly perform speculative execution to maximize performance [19] [20]. Assuming homogeneous environments, Hadoop default scheduler starts speculative tasks based on a naive heuristic that compares each task's progress to the average task progress of a job. LATE [21] MapReduce scheduling algorithm takes heterogeneous environments into consideration. However, LATE has a poor performance due to its static fixed-weight based method. SAMR algorithm [22] shares a similar idea as LATE. It, however, also leverages historical information to tune weights of map and reduce stages in order to get a more accurate estimation of task execution time. SAMR falls short of solving one crucial problem. It considers only hardware heterogeneity but not other factors, such as different job types and different job sizes, which also affect stage weights.

To overcome the deficiency of SAMR, we have developed ESAMR: an Enhanced Self-Adaptive MapReduce scheduling

1521-9097/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPADS.2012.30

IEEE computer society

algorithm. Like SAMR, ESAMR is inspired by the fact that slow tasks prolong the execution time of the whole job and different amounts of time are needed to complete the same task on different nodes due to hardware heterogeneity. In addition, considering that there are other factors that affect a task's progress [23], ESAMR records historical information for each node and adopts a k-means cluster identification algorithm to dynamically tune stage weight parameters and find slow tasks accurately. As a result, ESAMR significantly improves the performance of MapReduce scheduling in terms of estimating task execution time and launching backup tasks.

II. Background

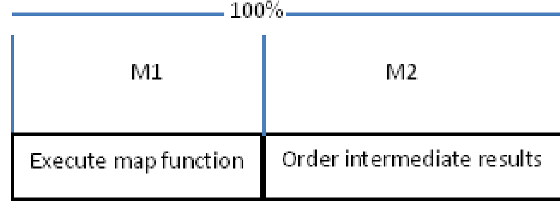
In this section, we first describe the MapReduce programming model. It is the basis of ESAMR scheduling algorithm. We also introduce Hadoop default scheduler, LATE scheduler, and SAMR scheduler. The deficiencies of these schedulers have motivated us to develop ESAMR scheduling algorithm.

A. Basic Concepts in MapReduce

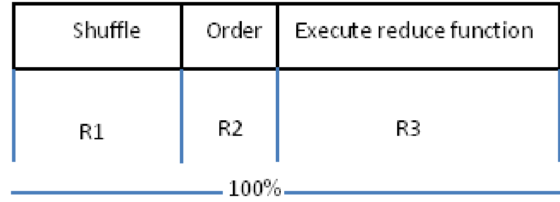
Next, we briefly describe how a Hadoop cluster works since other MapReduce-style clusters work similarly. In later parts of this paper, we will use the terms "Hadoop cluster" and "MapReduce cluster" interchangeably. A Hadoop cluster is often composed of many commodity PCs, where one PC acts as the master node and others as slave/worker nodes. A Hadoop cluster uses Hadoop Distributed File System (HDFS) [24] 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 [2] computation is comprised of two parts, 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 a Hadoop 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 worker nodes to execute map and reduce tasks. A worker node who is assigned a map task reads the content 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 sorted, written to the local disk, and divided into R regions by the partitioning 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 with certain keys. Thus, it has to retrieve its partition of data from all worker nodes that have executed the M map tasks. This process is called shuffle, which involves many-to-many communications among worker nodes. When a reduce task has read all intermediate data, it sorts it by the intermediate key so that all occurrences of the same key are grouped together. The reduce task then invokes the reduce function to produce the final output data (i.e., output key/value pairs) for its reduce partition [2]. Thus, a map task is often considered to have two stages: map and sort phases, while a reduce task has three stages: shuffle phase, sort phase, and reduce phase. Figure 1 illustrates the task stages and their time weights, i.e., $M1$, $M2$, $R1$, $R2$, and $R3$, where $M1 + M2 = 1$ and $R1 + R2 + R3 = 1$.

Fig. 1. MapReduce task stages



(a) Map Task



(b) Reduce Task

For a MapReduce job, all its map tasks are independent of each other and can be executed simultaneously. While reduce tasks depend on map tasks because the latter's outputs are the former's inputs, all reduce tasks are independent of each other and can be executed in parallel. Thus, the completion time of a map stage is determined by the slowest map task and a job's completion time (i.e., the completion time of a reduce stage) is determined by the slowest reduce task. To prevent slow tasks from prolonging the job execution time, Hadoop default scheduler, LATE scheduler, SAMR scheduler, and our ESAMR scheduler all employs a mechanism to speculatively re-execute slow tasks. Their speculative re-execution mechanisms are, however, different.

B. Existing Re-Execution Mechanisms

To select tasks for speculative re-execution, Hadoop default scheduler monitors the progress of tasks using a Progress Score (PS) between 0 and 1. The average progress score of a job is denoted by PS_{avg}. The progress score of the ith task is denoted by PS[i]. Suppose: a job has K number of tasks being executed; a task has a total of N number of key/value pairs to be processed and M of them have been processed successfully. Hadoop default scheduler gets PS according to Eq. (1) and Eq. (2), and then launches backup tasks according to Eq. (3).

$$PS = \begin{cases} M/N & \text{For Map tasks} \\ 1/3 * (K + M/N) & \text{For Reduce tasks} \end{cases} \quad (1)$$

$$PS_{avg} = \sum_{i=1}^{K} PS[i]/K \quad (2)$$

$$\text{For task } T_i\text{: } PS[i] < PS_{avg} - 20\% \quad (3)$$

Here, it is assumed that a map task spends negligible time in the order stage (i.e., M1=1 and M2=0) and a reduce task has finished K stages and each stage takes the same amount of time (i.e., R1=R2=R3=1/3). If Eq.(3) is satisfied, task T_i needs a backup task.

This method has several deficiencies, as pointed out by Chen et al. [22]. First, it uses fixed stage time weights by setting $M1$, $M2$, $R1$, $R2$, and $R3$ at 1, 0, 1/3, 1/3, and 1/3 respectively. The values of $M1$, $M2$, $R1$, $R2$, and $R3$ could, however, be different when tasks are running on different nodes, especially in a heterogeneous environment. Second, this scheduler launches backup tasks based on progress score (PS), which may not be appropriate, since in a heterogeneous environment, a lower PS does not necessarily mean a longer remaining execution time. Third, backup could be launched for fast tasks due to the inappropriate setting of stage weights. For instance, the first phase of a reduce task, i.e., the shuffle phase is often much slower than the sort and reduce phases, because shuffle involves communication with several map tasks over the network. Thus, a reduce task executing in the end of the shuffle phase could be a fast task. However, a backup may be launched for this task since its progress score, as computed by this method, is less than 1/3.

By launching backup for tasks with long remaining execution time, Longest Approximate Time to End (LATE) MapReduce scheduling algorithm overcomes the second deficiency of the aforementioned re-execution mechanism. LATE also first uses Eq.(1) to calculate a task's progress score (PS). But, it then computes a task's remaining execution time (denoted by TimeToEnd) with the following equations:

$$PR = PS/Tr \quad (4)$$

$$TimeToEnd = (1 - PS)/PR \quad (5)$$

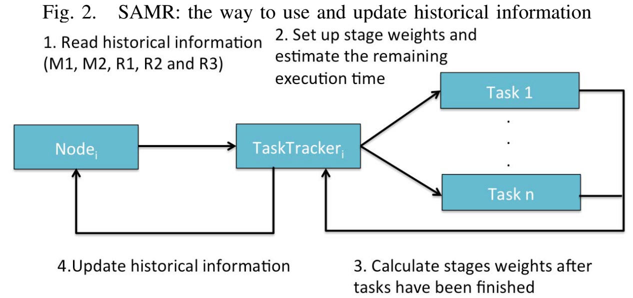
where Tr denotes how long a task T has executed and PR the task progress rate. TimeToEnd, how much time remains until T finishes, is estimated by Eq.(5).

Although LATE uses an improved strategy to launch backup tasks, it still frequently chooses wrong tasks to re-execute. This is because when estimating TimeToEnd, LATE still sets the stage time weights, i.e., M1, M2, R1, R2, and R3, at fixed values of 1, 0, 1/3, 1/3, and 1/3 respectively. This setting will lead to inaccurate TimeToEnd estimation.

Same as LATE algorithm, Self-Adaptive MapReduce (SAMR) scheduler identifies slow tasks via estimating task execution time. SAMR, however, does not use fixed stage weights for map and reduce tasks. It stores historical information of stage weight values on every node and updates it after every task execution on the node. When estimating the TimeToEnd of a task running on a node, SAMR reads historical information stored on the node to dynamically set the stage weights.

Figure 2 illustrates the way adopted by SAMR to use and update historical information on a node.

Fig. 2. SAMR: the way to use and update historical information



Since SAMR uses historical information recorded on each node to tune the stage weights, it adopts more realistic weights and could apply different weights for tasks running on different nodes. Thus, compared to Hadoop default and LATE schedulers, SAMR scheduler works better, especially in a heterogeneous environment.

SAMR only considers one factor that affects task stage weights, i.e., the hardware heterogeneity. However, even running on the same node, tasks of different MapReduce jobs could have different stage weights since it may take them different amounts of time to execute map and reduce functions and their amounts of intermediate data could be different. In addition, tasks of the same MapReduce job can have different stage weights when handling datasets of different sizes. For instance, larger input data size results in larger intermediate data size, which could cause relatively even more time spent on the shuffle phase.

This paper develops ESAMR: an Enhanced Self-Adaptive MapReduce scheduling algorithm. Knowing besides hardware heterogeneity there are many other factors that affect a task's stage weights, ESAMR records historical information on every node and adopts a k-means cluster identification algorithm to dynamically tune stage weights, estimate task execution time, and find slow tasks. Next section presents ESAMR algorithm in detail.

III. ESAMR ALGORITHM

In this section, we describe our new Enhanced Self-adaptive MapReduce (ESAMR) scheduling algorithm. ESAMR classifies the historical information stored on every node into k clusters using a machine learning technique. If a running job has completed some map tasks on a node, ESAMR records the job's temporary map phase weight (i.e., $M1$) on the node according to the job's map tasks completed on the node. The

temporary $M1$ weight is used to find the cluster whose $M1$ weight is the closest. ESAMR then uses the cluster's stage weights to estimate the job's map tasks' TimeToEnd on the node and identify slow tasks that need to be re-executed. If a running job has not completed any map task on a node, the average of all k clusters' stage weights are used for the job. In the reduce stage, ESAMR carries out a similar procedure. After a job has finished, ESAMR calculates the job's stage weights on every node and saves these new weighs as a part of the historical information. Finally, ESAMR applies k-means, a machine learning algorithm, to re-classify the historical information stored on every worker node into k clusters and saves the updated average stage weights for each of the k clusters. By utilizing more accurate stage weights to estimate the TimeToEnd of running tasks, ESAMR can identify slow tasks more accurately than SAMR, LATE, and Hadoop default scheduling algorithms. Algorithm 1 gives the pseudo code of ESAMR algorithm.

Algorithm 1 ESAMR

Require:

1: PFM (Percentage of Finished Map Tasks), a threshold used to control when to begin the slow map task identification
2: PFR (Percentage of Finished Reduce Tasks), a threshold used to control when to begin the slow reduce task identification
3: $history$, historical information of the k clusters, where each record of a cluster contains 5 values, $M1$, $M2$, $R1$, $R2$ and $R3$
4: **if** a job has completed PFM of its map tasks **then**
5: $M1$= **CalculateWeightsMapTasks**
6: $M2$=1-$M1$
7: **end if**
8: **if** a job has completed PFR of its reduce tasks **then**
9: $< R1, R2 >$= **CalculateWeightsReduceTasks**
10: $R3$=1-$R1$-$R2$
11: **end if**
12: $slowTasks$= **FindSlowTasks**
13: run backup tasks for $slowTasks$
14: **if** a job has finished **then**
15: run k-means algorithm to re-classify historical information into k clusters
16: **end if**

In statistics and data mining, k-means [25] [26] clustering is a method of cluster analysis. The main purpose of k-means clustering is to partition a set of entities into different clusters in which each observation belongs to a cluster with the nearest mean value.

In our k-means algorithm, ESAMR first assigns random values for the centroids (i.e., mean values) of k groups. Second, ESAMR assigns each entity to a cluster that has the closest centroid. Third, ESAMR recalculates the centroids and repeats the second and third steps until entities can no longer change groups.

Algorithm 2 CalculateWeightsMapTasks

1: **if** a node has finished map tasks for the job **then**
2: calculate $tempM1$ based on the job's map tasks completed on the node
3: $M1$=randomly chosen first stage weight $M1$ from the corresponding node's $history$
4: $beta$=abs($tempM1$-$M1$)
5: **for** each $M1[i] \in the\ node's\ history$, i=1,2,...,k **do**
6: **if** abs($M1[i]$-$tempM1$)$<beta$ **then**
7: $M1$=$M1[i]$
8: $beta$=abs($tempM1$-$M1$)
9: **end if**
10: **end for**
11: return $M1$
12: **else**
13: $M1$=$\sum_{i=1}^{k} M1[i]/k$
14: return $M1$
15: **end if**

Algorithm 3 CalculateWeightsReduceTasks

1: **if** a node has finished reduce tasks for the job **then**
2: calculate $tempR1$ based on the job's reduce tasks completed on the node
3: calculate $tempR2$ based on the job's reduce tasks completed on the node
4: $< R1, R2 >$=a randomly chosen $R1$ and $R2$ pair from the node's $history$
5: $beta = abs(tempR1 - R1) + abs(tempR2 - R2)$
6: **for** each $R1[i]$ and $R2[i]$ pair in the node's $history$, i=1,2,..,k **do**
7: **if** abs($tempR1$-$R1[i]$)+abs($tempR2$-$R2[i]$)$<beta$ **then**
8: $R1$=$R1[i]$
9: $R2$=$R2[i]$
10: $beta$=abs($tempR1$-$R1$)+abs($tempR2$-$R2$)
11: **end if**
12: **end for**
13: return$< R1, R2 >$
14: **else**
15: $R1$=$\sum_{i=1}^{k} R1[i]/k$
16: $R2$=$\sum_{i=1}^{k} R2[i]/k$
17: return$< R1, R2 >$
18: **end if**

Each worker node runs k-means algorithm to classify the historical information stored on the node. No additional communication between nodes is needed when reading and updating historical information. According to our experiments, the running time of the k-means algorithm is around 80

milliseconds on a node. Since it only runs once upon the completion of a MapReduce job, we think it adds a negligible overhead to ESAMR algorithm. Algorithm 6 gives the pseudo code of the k-means algorithm used in ESAMR.

Algorithm 4 FindSlowTasks

```
1: STT (Slow Task Threshold), a variable for selecting slow
   tasks
2: set SlowTasks //a temp list to save all slow tasks
3: for each job that has completed PFM (or PFR) of its
   map (or reduce) tasks do
4:   for each running task i of the job do
5:     PS_i=CalculateProgressScore
6:     PR_i=PS_i/Tr_i, where Tr_i is the time that has been
   used by the task
7:     TTE_i= (1-PS_i)/PR_i
8:   end for
9:   ATTE=Σ_{i=1}^{N} TTE_i/N, where N is the total number of
   running tasks of the job
10:  for each running task i of the job do
11:    if TTE_i-ATTE > ATTE * STT then
12:      slowTasks.add(ith task)
13:    end if
14:  end for
15: end for
16: return SlowTasks
```

Algorithm 5 CalculateProgressScore

```
1: SubPS=N_f/N_a, where N_f is the number of key/value
   pairs which have been processed in a phase of a task and
   N_a is the total number of key/value pairs that need to be
   processed in a phase of the task
2: if the task is a map task then
3:   if the map task is at the first phase then
4:     PS = M_1 * SubPS
5:   else
6:     PS = M_1+M_2 * SubPS
7:   end if
8: end if
9: if the task is a reduce task then
10:  if the reduce task is at the first phase then
11:    PS = R_1 * SubPS
12:  else if the reduce task is at the second phase then
13:    PS = R_1+R_2 * SubPS
14:  else
15:    PS = R_1+R_2+R_3 * SubPS
16:  end if
17: end if
18: return PS
```

Algorithm 6 K-means

```
Require: E=e_1,e_2,...,e_n (set of entities to be clustered)
1:     k (number of clusters)
2:     MaxIters (Maximum number of iterations)
3:
Ensure: C = {c_1,c_2,...,c_k} (set of cluster centroids)
4:   L = {l(e)|e = 1, 2, ..., n} (set of cluster labels of E)
5:
6: for i = 1 to k do
7:   c_i=e_j (randomly select an e_j from E)
8: end for
9: for e_i ∈ E do
10:  l(e_i) = argminDistance(e_i, c_j), j ∈ {1...k}// find the
   cluster j whose center is nearest to an entity
11: end for
12: iter = 0
13: repeat
14:   for c_i ∈ C do
15:     c_i= avg(e_k), for all l(e_k) = i
16:   end for
17:   changed = false
18:   for e_i ∈ E do
19:     clusterID= argminDistance(e_i, c_j), j ∈ {1...k}
20:     if clusterID ≠ l(e_i) then
21:       l(e_i) = clusterID
22:       changed = true
23:     end if
24:   end for
25:   iter++
26: until changed = false or iter > MaxIters
```

IV. Evaluation

To evaluate our ESAMR algorithm, we compare it with the other two existing algorithms designed to work in heterogeneous environments: SAMR and LATE algorithms. Since in [22] Chen et al. have shown that SAMR is advantageous than Hadoop default scheduler, we only compare ESAMR with SAMR and skip the comparison with Hadoop default scheduler. We modified Hadoop-0.21 and integrated each one of the aforementioned three re-execution algorithms into it. We run WordCount and Sort jobs, which are classic examples of Hadoop applications, to evaluate the algorithms' performance.

Three metrics: weight estimation error, TimeToEnd estimation error, and identified slow tasks, are used for evaluation. We run experiments in a cluster of 1 master node and 5 worker nodes that are configured as a rack. Table I lists the cluster hardware environment and configuration. For ESAMR algorithm, we set its parameters as follows: PFM at 20%, PFR at 20%, k at 10, and STT at 40%. The same PFM, PFR, and STT parameter values are used for SAMR and LATE algorithms. In addition, we have carefully tuned SAMR and LATE algorithms. For instance, HISTORY_PRO(HP) is an important parameter for SAMR [22]. We set HP at 0.2 since that value has been shown by experiments to achieve the best performance [22] for SAMR.

TABLE I
EVALUATION ENVIRONMENT

Nodes	Quantity	Hardware and Hadoop Configuration
Master node	1	2 single-core 2.2GHz Optron-64 CPUs, 6GB RAM, 1Gbps Ethernet
Type-A worker nodes	3	2 single-core 2.2GHz Optron-64 CPUs, 4GB RAM, 1Gbps Ethernet, 2 map and 1 reduce slots per node
Type-B worker nodes	2	2 single-core 2.3GHz Optron-64 CPUs, 2GB RAM, 100Mbps Ethernet, 2 map and 1 reduce slots per node

To verify the correctness of estimation, we list the stage weights estimated by ESAMR and the actual weights collected from the system in Table II. Because M1+M2=1 and R1+R2+R3=1, we only list M1, R1 and R3 to show the result. From Table II, we see that weights estimated by ESAMR are not far from the real weights. However, all stage weights are far from the constant weights (1, 0, 1/3, 1/3, 1/3) used in LATE algorithm. Comparing data in Tables II and III, we conclude that the weight estimation errors of SAMR are bigger than those of ESAMR.

TABLE II
WEIGHTS ESTIMATED BY ESAMR VS REAL WEIGHTS OF A WORDCOUNT 10GB JOB

Node Name	M1	R1	R3
Node 1	0.7261/0.7198	0.1926/0.1901	0.8062/0.8078
Node 2	0.7633/0.7502	0.1917/0.1899	0.8072/0.8090
Node 3	0.6200/0.6109	0.2060/0.2079	0.7920/0.7814
Node 4	0.2142/0.2078	0.3647/0.3699	0.6327/0.6284
Node 5	0.2062/0.2012	0.3954/0.3894	0.6028/0.6012

TABLE III
WEIGHTS ESTIMATED BY SAMR VS REAL WEIGHTS OF A WORDCOUNT 10GB JOB

Node Name	M1	R1	R3
Node 1	0.9563/0.7902	0.5717/0.2247	0.4248/0.7747
Node 2	0.2942/0.8074	0.5839/0.2332	0.4116/0.7661
Node 3	0.9487/0.7836	0.5683/0.1794	0.4276/0.8197
Node 4	0.8241/0.5922	0.4990/0.3395	0.4513/0.5549
Node 5	0.8164/0.4071	0.6949/0.2960	0.2916/0.6948

Next, we compare the TimeToEnd estimation error of the three algorithms. Figures 3 and 4 show the TimeToEnd estimation error of map and reduce tasks by ESAMR, SAMR, and LATE on a WordCount 10GB job. The job has 100 map tasks. For convenience, we chose the first 20 map tasks to show the performance, since 20 map tasks are enough to illustrate the difference. There are 20 reduce tasks in the job. We use all 20 reduce tasks to show the algorithm performance. Figures 3 and 4 illustrate the effectiveness of ESAMR. Among the three algorithms, ESAMR always leads to the smallest prediction error. With ESAMR, the differences between estimated and actual TimeToEnd of map and reduce tasks average at 4 and 5 seconds respectively. With SAMR, the differences average at 38 and 27 seconds respectively and with LATE, the average differences are 64 and 129 seconds respectively.

Fig. 3. Map task TimeToEnd estimation error (WordCount 10GB)

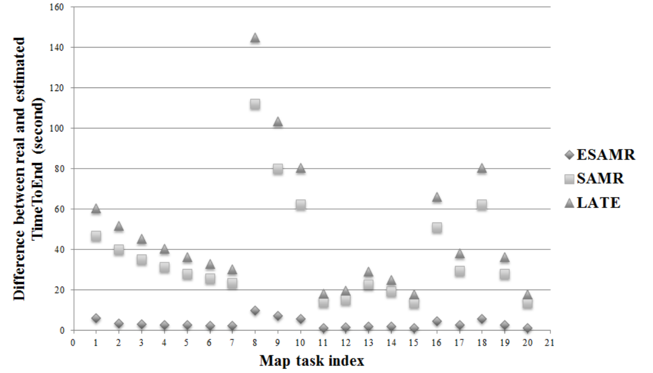
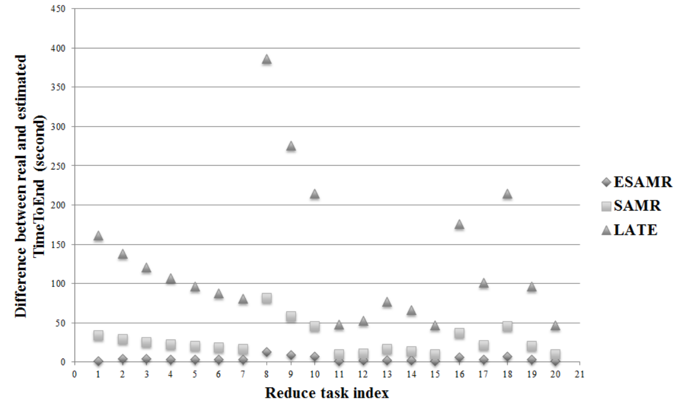


Fig. 4. Reduce task TimeToEnd estimation error (WordCount 10GB)



Figures 5 and 6 show the TimeToEnd estimation error of map and reduce tasks by ESAMR, SAMR and LATE on a Sort 10GB job. From these figures, we can see that ESAMR still has the smallest error, but initially (i.e., for the first 7 tasks) LATE has a better performance than SAMR. The reason is that the default stage weights used by LATE are closer to the real weights than those used by SAMR which are still based on the historical data collected from running the previous WordCount job. As SAMR begins to use historical information from running the Sort job to adjust the stage weights, it performs better than LATE for the second half of the experiment (i.e., from the 8th to the 20th tasks). With ESAMR, the differences between estimated and actual TimeToEnd of map and reduce tasks average at 0.77 and 3 seconds respectively. With SAMR, the average differences are 14 and 83 seconds respectively and with LATE, the average differences are 27 and 139 seconds respectively.

Figure 7 shows the map task execution time estimated by ESAMR, SAMR and LATE vs. the real execution time. From the figure, we can see that ESAMR considers the 8th and 9th map tasks as the slow tasks. SAMR chooses the 10th map task and LATE chooses the 1st map task as the slow task. Based on

Fig. 5. Map task TimeToEnd estimation error (Sort 10GB)

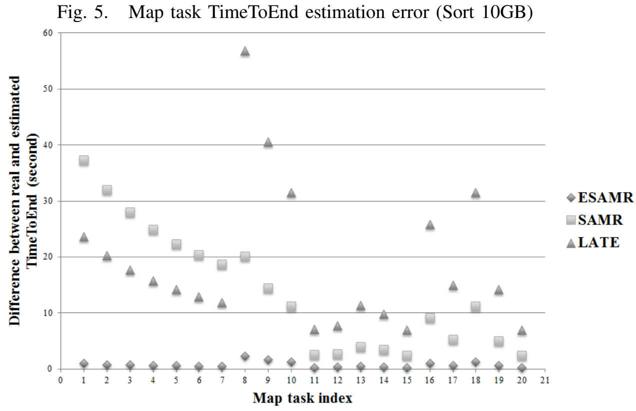
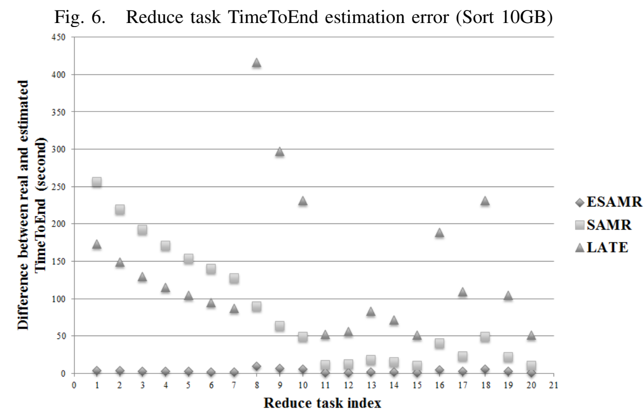


Fig. 6. Reduce task TimeToEnd estimation error (Sort 10GB)



the real execution time, we know the slow tasks are actually the 8th and 9th map tasks. Only ESAMR has identified the slow tasks correctly.

Figure 8 shows the reduce task execution time estimated by ESAMR, SAMR and LATE vs. the real execution time. ESAMR estimates the 1st reduce task as the slow task, while SAMR chooses the 8th reduce task and LATE chooses the 8th and 9th reduce tasks. The slow reduce task is actually the 1st task. ESAMR is the only algorithm that has identified the slow reduce task correctly.

V. Conclusion

To overcome the limitations of existing MapReduce re-execution mechanisms, in this paper we develop ESAMR: an Enhanced Self-Adaptive MapReduce scheduling algorithm, which uses k-means clustering algorithm to classify historical information into k clusters and thus generates more accurate estimation of task's stage weights to correctly identify slow tasks and re-execute them. Experimental results have shown the effectiveness of ESAMR.

Fig. 7. Real and estimated map task execution time of a WordCount 10GB job

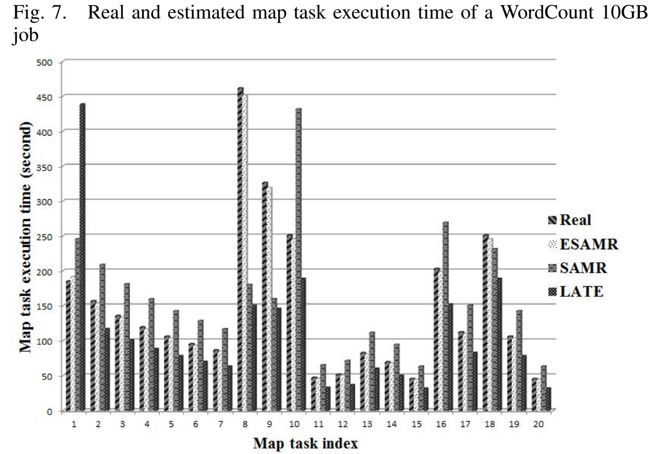
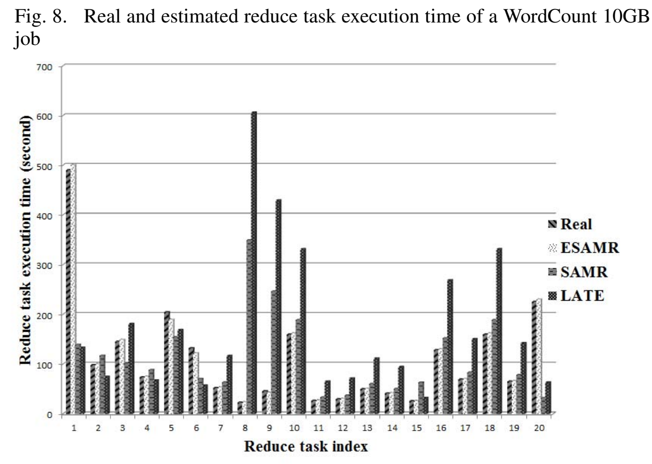


Fig. 8. Real and estimated reduce task execution time of a WordCount 10GB job



Acknowledgements

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

References

[1] P. Dubey, "A platform 2015 workload model recognition, mining and synthesis moves computers to the era of tera," *White paper, Intel Corporation*, 2008.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of the 21st ACM SIGOPS symposium on Operating systems principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, (Berkeley, CA, USA), pp. 15–15, USENIX Association, 2006.

[5] P. R. Elespuru, S. Shakya, and S. Mishra, “Mapreduce system over heterogeneous mobile devices,” in *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS ’09, (Berlin, Heidelberg), pp. 168–179, Springer-Verlag, 2009.

[6] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos, “Supporting mapreduce on large-scale asymmetric multi-core clusters,” *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 25–34, April 2009.

[7] M. de Kruijf and K. Sankaralingam, “MapReduce for the Cell B.E. Architecture,” Tech. Rep. TR1625, Department of Computer Sciences, The University of Wisconsin-Madison, Madison, WI, 2007.

[8] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, “Mars: a mapreduce framework on graphics processors,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT ’08, (New York, NY, USA), pp. 260–269, ACM, 2008.

[9] M. C. Schatz, “Cloudburst: highly sensitive read mapping with mapreduce,” *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009.

[10] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, “Spatial queries evaluation with mapreduce,” in *Proceedings of the 8th International Conference on Grid and Cooperative Computing*, GCC ’09, (Washington, DC, USA), pp. 287–292, IEEE Computer Society, 2009.

[11] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Job scheduling for multi-user mapreduce clusters,” Tech. Rep. UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[12] C. Tian, H. Zhou, Y. He, and L. Zha, “A dynamic mapreduce scheduler for heterogeneous workloads,” in *Proceedings of the 8th International Conference on Grid and Cooperative Computing*, GCC ’09, (Washington, DC, USA), pp. 218–224, IEEE Computer Society, 2009.

[13] C. Jin and R. Buyya, “Mapreduce programming model for .net-based cloud computing,” in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par ’09, (Berlin, Heidelberg), pp. 417–428, Springer-Verlag, 2009.

[14] “Hadoop.” http://hadoop.apache.org.

[15] “Apache hadoop.” http://en.wikipedia.org/wiki/Apache_Hadoop.

[16] “Yahoo hadoop tutorial.” http://public.yahoo.com/gogate/hadoop-tutorial/starttutorial.html.

[17] S. Manoharan, “Effect of task duplication on the assignment of dependency graphs,” *Parallel Comput.*, vol. 27, pp. 257–268, February 2001.

[18] “Hbase.” http://hbase.apache.org/book/book.html.

[19] G. Barish, “Speculative plan execution for information agents,” tech. rep., University of Southern California, 2003.

[20] M. Kafil and I. Ahmad, “Optimal task assignment in heterogeneous computing systems,” in *Proceedings of the 6th Heterogeneous Computing Workshop*, HCW ’97, (Washington, DC, USA), pp. 135–, IEEE Computer Society, 1997.

[21] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI’08, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[22] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, “Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment,” in *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*, CIT ’10, (Washington, DC, USA), pp. 2736–2743, IEEE Computer Society, 2010.

[23] R. Nanduri, N. Maheshwari, A. Reddyraja, and V. Varma, “Job aware scheduling algorithm for mapreduce framework,” in *Proceedings of the 3rd International Conference on Cloud Computing Technology and Science*, CLOUDCOM ’11, (Washington, DC, USA), pp. 724–729, IEEE Computer Society, 2011.

[24] “Hdfs.” http://hadoop.apache.org/common/docs/current/hdfsdesign.html.

[25] “K-means.” http://en.wikipedia.org/wiki/K-means_clustering.

[26] G. Hamerly and C. Elkan, “Alternatives to the k-means algorithm that find better clusterings,” in *Proceedings of the 11th international conference on Information and knowledge management*, CIKM ’02, (New York, NY, USA), pp. 600–607, ACM, 2002.