HPMR: Prefetching and Pre-shuffling in Shared MapReduce Computation Environment

Sangwon Seo[1], Ingook Jang[1], Kyungchang Woo[2], Inkyo Kim[3], Jin-Soo Kim[4], and Seungryoul Maeng[1]

[1]*Computer Science Department*
Korea Advanced Institute of Science
and Technology (KAIST), South Korea
{sangwon.seo,jik,maeng}@kaist.ac.kr

[2]*Search Engineering*
Yahoo! Inc, Seoul
South Korea
kc@yahoo-inc.com

[3]*Advanced Software Center*
Samsung Advanced Institute
of Technology, South Korea
inkyo.kim@samsung.com

[4]*School of Info. and Comm.*
Sungkyunkwan University
South Korea
jinsookim@skku.edu

***Abstract*—MapReduce is a programming model that supports distributed and parallel processing for large-scale data-intensive applications such as machine learning, data mining, and scientific simulation. Hadoop is an open-source implementation of the MapReduce programming model. Hadoop is used by many companies including Yahoo!, Amazon, and Facebook to perform various data mining on large-scale data sets such as user search logs and visit logs. In these cases, it is very common to share the same computing resources by multiple users due to practical considerations about cost, system utilization, and manageability. However, Hadoop assumes that all cluster nodes are dedicated to a single user, failing to guarantee high performance in the shared MapReduce computation environment.**

In this paper, we propose two optimization schemes, *prefetching* and *pre-shuffling*, which improve the overall performance under the shared environment while retaining compatibility with the native Hadoop. The proposed schemes are implemented in the native Hadoop-0.18.3 as a plug-in component called HPMR (High Performance MapReduce Engine). Our evaluation on the Yahoo!Grid platform with three different workloads and seven types of test sets from Yahoo! shows that HPMR reduces the execution time by up to 73%.

I. Introduction

Internet services, such as search engines, on-line portals, e-commerce sites, and social networking sites, have recently emerged as an important class of computer applications, delivering their contents to more than millions of users. These services not only deal with enormous volumes of data, but also generate a large amount of data which needs to be processed every day. MapReduce is a programming model that supports distributed and parallel processing for large-scale data-intensive applications such as machine learning, data mining, and scientific simulation. MapReduce divides a computation into multiple small tasks and let them run on different machines in parallel. It is highly scalable because thousands of commodity machines can be used as an effective platform for distributed computing. In addition, the MapReduce programming model is designed to be accessible to the widest possible class of developers; it favors simplicity at the expense of generality by hiding implementation details and providing only simple abstract APIs.

Hadoop [3] is an open-source implementation of the MapReduce programming model. It is originally developed by Yahoo!, but is also used by other companies including Amazon, Facebook, and The New York Times due to high performance, reliability, and availability that it offers. Hadoop relies on its own distributed file system called HDFS (Hadoop Distributed File System), which is a mimic of GFS (Google File System). Like GFS, HDFS has a master/slave architecture; an HDFS cluster consists of a single NameNode and a number of DataNodes. NameNode is the master server that manages the namespace of a file system and regulates clients' access to files, while DataNodes manage storage directly attached to each DataNode. HDFS has a coarse-grained placement policy of replicas. With the replication factor of three, the common placement policy of HDFS is to place one replica on one node in the local rack, the second on a different node in the local rack, and the last on a node in a different rack. This policy generally improves write performance by cutting down inter-rack write traffic.

The general architecture and the typical workflow of MapReduce are illustrated in Figure 1. An input file is saved in HDFS and a part of input file, which we call an *input split*, is disseminated to the corresponding map task. The size of the input split is limited to the block size, and each task is mapped to only one input split. The output of the map task, called *intermediate output*, is sent to the *combiner* which pre-reduces the intermediate output in a local node. The combiner aggregates multiple intermediate outputs generated from the node to a single large intermediate output to reduce network overhead. In this way, each local node produces only one intermediate output. This combined intermediate output is passed to the *partitioner*, while key-value pairs are shuffled to the corresponding reduce task over the network. The partitioner determines to which partition a given key-value pair will go. The default partitioner computes a hash value for the key and assigns the partition based on this result. Therefore, the key-value pairs with the same key are shuffled to the same reduce task for sorting and reducing.

The shuffling overhead increases as the degree of parallel processing increases. The larger the number of map tasks, the longer the shuffling phase takes to complete. The network bandwidth between nodes is also an important factor of the shuffling overhead. Thus, it is essential to reduce the shuffling overhead to improve the overall performance of the MapReduce computation.

978-1-4244-5012-1/09/$25.00 ©2009 IEEE

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on December 11, 2009 at 10:32 from IEEE Xplore. Restrictions apply.

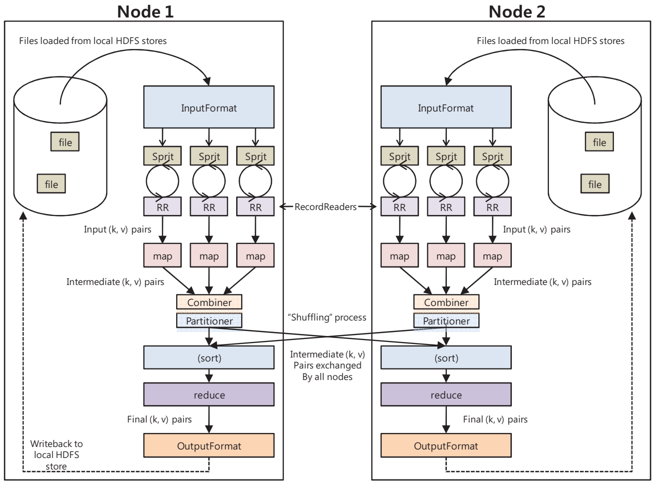


Fig. 1. The architecture of MapReduce using Hadoop.

One of Hadoop's basic principles is "moving computation is cheaper than moving data." This principle indicates that it is often better to migrate the computation closer to where the data is located rather than to move data where the application is running. This is especially true when the size of the data set is huge because the migration of the computation minimizes network congestion and increases the overall throughput of the system. When a computation task is located near the data it consumes, we call that the task has good *data locality*. The best data locality can be achieved if the needed data and the computation task are placed in the same node. The next best case is when the data is in any other node within the same rack.

Hadoop assumes that cluster nodes are dedicated to the MapReduce computation [8]. In Yahoo!, however, a large experimental cluster called Yahoo!Grid is managed by Hadoop-On-Demand (HOD) [4]. HOD is a management system for provisioning virtual Hadoop clusters over a large physical cluster. Each engineer at Yahoo! has a log-on account for the cluster and allocates his or her own virtual nodes on demand from a pool of physical nodes. Therefore, all physical nodes are shared by more than one Yahoo! engineers.

HOD increases the utilization of physical resources at the expense of performance. When the underlying computing resources are shared by multiple users, the current Hadoop's policy of "moving computation where the data is located" may not be effective at all. For example, imagine a situation where a large log file is saved across the entire Hadoop nodes. If a Yahoo! engineer is allowed to use only 10% of the total nodes to process the log file, 90% of the input data still should be read from other nodes. Moreover, since many users are competing for network and hardware resources, the performance will be deteriorated even more significantly compared to the dedicated MapReduce environment. This problem motivated us to develop innovative ways for improving performance under the shared MapReduce environment.

In this paper, we propose two optimization schemes, *prefetching* and *pre-shuffling*, to overcome the aforementioned problem for the shared MapReduce environment. These schemes are implemented in our High Performance MapReduce Engine (HPMR). The prefetching scheme can be broadly classified into two types: *the intra-block prefetching* and *the inter-block prefetching*. In the intra-block prefetching, only an input split or an intermediate output is prefetched, while the whole candidate data block is prefetched in the inter-block prefetching. These types of prefetching are used for all map and reduce phases. The pre-shuffling scheme reduces the amount of intermediate output to shuffle. During pre-shuffling, HPMR looks over an input split before the map phase begins and predicts the target reducer where the key-value pairs are partitioned. As illustrated in Figure 1, if key-value pairs of intermediate output are partitioned into a local node, the number of shuffling operations over the network can be reduced. We have designed a new task scheduler for pre-shuffling, and the pre-shuffling scheme is used only for the reduce phase. In brief, the prefetching scheme improves the data locality, and the pre-shuffling scheme significantly reduces the shuffling overhead during the reduce phase. We have modified the native Hadoop version 0.18.3 to build HPMR and evaluated its performance on a real Yahoo!Grid platform. We summarize contributions of our paper as follows.

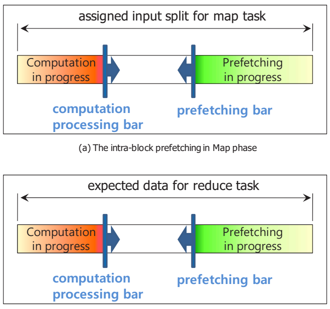
- We have analyzed performance degradation of Hadoop in depth in a shared MapReduce computation environment.
- We have designed new prefetching and pre-shuffling schemes, which significantly improve the MapReduce performance especially when physical nodes are shared by multiple users.
- We have built the High Performance MapReduce Engine (HPMR) that exploits data locality and reduces network overhead. It is implemented as a plug-in component for the native Hadoop system and is compatible with both dedicated and shared environments.
- Compared with the native Hadoop, our implementation of HPMR has demonstrated the excellent performance on various platforms ranging from a single multi-core node to clusters with thousands of nodes. Especially, we have shown that the execution times of complex MapReduce applications such as ad-hoc analysis of very large data sets, data mining, etc., are reduced significantly in the shared MapReduce environment. These applications are common in major Internet companies such as Yahoo!

The rest of this paper is organized as follows. Some of the related work are briefly described in the next section. The design and the implementation of HPMR are described in Section III and Section IV, respectively. In these sections, we discuss several issues and suggest our solutions in detail. Section V presents our experimental evaluation results obtained from the Yahoo!Grid platform. Finally, we conclude the paper in Section VI.

II. RELATED WORK

HPMR is related to a broad class of prior literature ranging from MapReduce [1] to traditional prefetching techniques.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on December 11, 2009 at 10:32 from IEEE Xplore. Restrictions apply.



(a) The intra-block prefetching in Map phase

(b) The intra-block prefetching in Reduce phase

Fig. 2. The intra-block prefetching

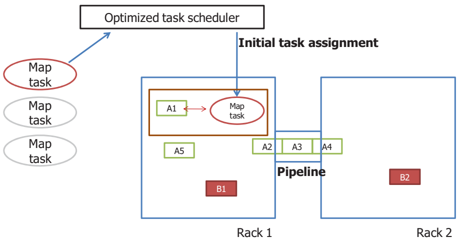


Fig. 3. An example of inter-block prefetching. The task scheduler assigns a map task while prefetching the required blocks (A2, A3, and A4) in a pipeline manner.

Considerable work has been carried out on prefetching methods to reduce I/O latency [16][18][19]. Most of their work focuses on file prefetching in local file system, but our work focuses on distributed data prefetching in distributed file system for maximizing network bandwidth and minimizing I/O latency. Our prefetching scheme is inspired by predictive prefetching for World Wide Web [16], which aimed at reducing the web latency perceived by users.

Zaharia et al. [8] have proposed a new scheduling algorithm called LATE (Longest Approximation Time to End), which finds stragglers (i.e., tasks that take unusually long time to complete) more efficiently in the shared environment. They showed that the Hadoop's current scheduler can cause severe performance degradation in heterogeneous environments such as virtualized data centers where uncontrollable variations in performance exist.

Dryad [9] from Microsoft is more flexible than MapReduce, as it allows for the execution of arbitrary computation that can be expressed as directed acyclic graph. As mentioned in Dryad [9], the degree of data locality is highly related to the MapReduce performance. In Yahoo!Grid, however, providing good data locality is difficult due to the presence of shared users.

III. DESIGN

In this section, we present the design of HPMR in detail. We describe the prefetching scheme in Section III-A, and the pre-shuffling scheme in Section III-B. The emphasis of this section is not on the syntactical details of HPMR, but on how the proposed scheme meets the design goals and features.

A. Prefetching Scheme

A computation requested by an application will be performed much more efficiently if it is executed near the data it operates on. As described in Section I, Yahoo!Grid is shared by multiple users using Hadoop-On-Demand (HOD). Dedicating a number of nodes to a single user is probably overkill. In practice, not only the cluster is shared, but also there is a limitation in the number of nodes a user can use. In this case, it is not easy to guarantee good data locality to all computation tasks.

HPMR provides a prefetching scheme to improve the degree of data locality. The prefetching scheme can be classified into two types: the intra-block prefetching and the inter-block prefetching.

1) Intra-block prefetching: The intra-block prefetching is a simple prefetching technique that prefetches data within a single block while performing a complex computation. As shown in Figure 2, it is executed in a bi-directional manner. While a complex job is performed in the left side, the to-be-required data are prefetched and assigned in parallel to the corresponding task.

The intra-block prefetching has a couple of issues that need to be addressed. First, it is necessary to synchronize computation with prefetching. We have solved this problem using the concept of processing bar that monitors the current status of each side and invokes a signal if synchronization is about to be broken. The processing bar not only works as a synchronization manager, but also provides a way to measure the effectiveness of this technique. The second issue is to find the proper prefetching rate. As generally believed, we first assumed that the more the prefetching is done, the better the performance gets. Unfortunately, the experimental results have shown that this is not always the case. Therefore, through repetitive experiments, we try to find the appropriate prefetching rate at which the performance can be maximized while minimizing the prefetching overhead.

The bi-directional processing provides several benefits. Most of all, it simplifies the implementation, allowing our plug-in interfaces to be used independently without directly modifying the native Hadoop. As shown in Figure 2, the bi-directional processing uses two types of processing bars, one for computation and the other for prefetching. These bars enable both of computation and prefetching to occur simultaneously, preventing the overlap of their processing. In this way, we not only simplify the implementation, but also solve the synchronization problem between computation and prefetching. In addition, the bi-directional processing helps to minimize the network overhead. In case the prefetching is always initiated from the beginning of a data block, the prefetcher will read the data repeatedly from the single source. If the perfetching rate is not fast enough to catch up the computation rate, it will waste network bandwidth without making any effect on the performance. However, our scheme successfully removes redundant read operations and minimizes

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on December 11, 2009 at 10:32 from IEEE Xplore. Restrictions apply.

I/O overhead in a bi-directional fashion.

In brief, the intra-block prefetching is simple, yet powerful for complex computations which require a large amount of data.

2) Inter-block prefetching: Recall that the previous intra-block prefetching runs within a single block. On the other hand, the inter-block prefetching runs in block level, by prefetching the expected block replica to a local rack. An example of inter-block prefetching is shown in Figure 3, where the data blocks, A2, A3, and A4, required by the map task are prefetched from the rack 2 to the rack 1 in a pipelined manner.

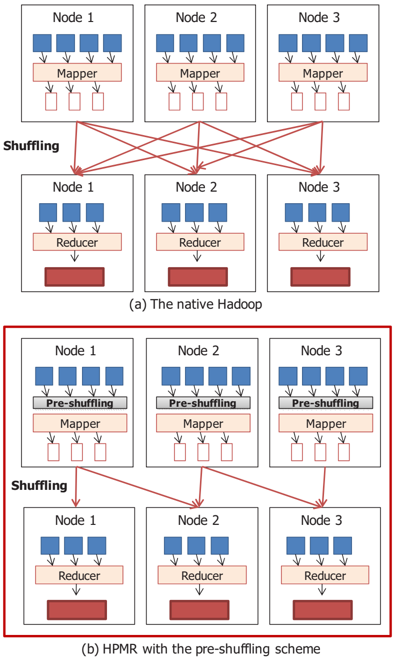
Algorithm 1 outlines the basic steps of the inter-block prefetching. Initially, our optimized task scheduler discussed in Section IV-A assigns map tasks to the nodes that are the nearest to the required blocks. And then, the predictor generates the list of data blocks, B, to be prefetched for the target task t. For each data block b in B, the prefetcher first identifies the locations of replicas for b. If the local rack does not have any replica of b and there is an overloaded replica whose access frequency is beyond the predefined threshold, the prefetcher tries to increase the number of replicas by replicating b to the local node. If the local node does not have enough space, it is replicated to one of nodes in the local rack. When replicating a data block by the inter-block prefetching, it is read from the least loaded replica so as not to minimize the impact on the overall performance.

Algorithm 1 The inter-block prefetching algorithm

```
INPUT: $t$ /* the target task */
$B \leftarrow$ CandidateBlocks ($t$); /* Get the candidate blocks for $t$ */
$<n, r> \leftarrow$ GetInfo ($t$); /* Get the node number ($n$) and the rack number ($r$) of the task $t$ */
for each $b \in B$ do
    if a replica of $b$ already exists in the local rack $r$ then
        continue;
    end if
    if there is an overloaded replica whose access frequency is larger than Threshold then
        $n_{src} \leftarrow$ the node number of the least loaded replica;
        if the node $n$ has enough disk space then
            Prefetch the data block $b$ from $n_{src}$ to $n$;
        else
            Choose a node $n'$ with enough space in the rack $r$;
            Prefetch the data block $b$ from $n_{src}$ to $n'$;
        end if
    end if
end for
```

B. Pre-shuffling Scheme

The pre-shuffling scheme in HPMR significantly reduces the amount of intermediate outputs to shuffle, as shown in Figure 4. Figure 4(a) shows the original MapReduce workflow, and Figure 4(b) illustrates the revised version of HPMR workflow with pre-shuffling. The key idea of pre-shuffling is quite simple; the pre-shuffling module in the task scheduler looks over input split or candidate data in the map phase, and predicts which reducer the key-value pairs are partitioned into. The expected data are assigned to a map task near the future reducer before the execution of the mapper. Note that if key-value pairs of intermediate output are partitioned into a local node, it will reduce the number of shuffling over the network. As shown in Figure 4, the use of pre-shuffling dramatically decreases the number of shuffling from $O(M^2)$ to $O(M)$, where M denotes the number of mappers.



(a) The native Hadoop

(b) HPMR with the pre-shuffling scheme

Fig. 4. The comparison between the native Hadoop and HPMR with pre-shuffling.

C. Optimization

MapReduce automatically handles failures without letting programmers know the details of fault-tolerance operations. When a node crashes, MapReduce reassigns the task to another live node. For faster computation, a straggler, a node performing poorly though available, invokes MapReduce to run a speculative copy of the currently running task on another machine. Google claims that the speculative execution improves the job response time by 44% [1]. However, it did not work well on our experimental cluster, Yahoo!Grid. Unlike the public release version of Hadoop, the option for the speculative execution is turned off in Yahoo!Grid. This is because Yahoo!Grid is a shared environment, while Hadoop assumes that cluster nodes are solely dedicated to a user [8].

The LATE algorithm actually aims to address the problem of how to robustly perform speculative execution to maximize performance under heterogenous environment. However, it did not consider data locality that can accelerate the MapReduce

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on December 11, 2009 at 10:32 from IEEE Xplore. Restrictions apply.

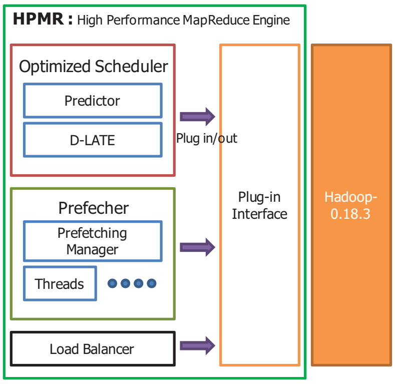


Fig. 5. The overall architecture of HPMR.

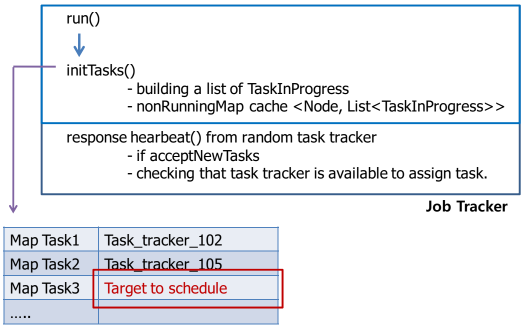

run()
initTasks()
- building a list of TaskInProgress
- nonRunningMap cache <Node, List<TaskInProgress>>
response hearbeat() from random task tracker
- if acceptNewTasks
- checking that task tracker is available to assign task.
Job Tracker
Map Task1
Task_tracker_102
Map Task2
Task_tracker_105
Map Task3
Target to schedule
.....

Fig. 6. The native Hadoop scheduler.

computation further. To complement the LATE algorithm, we propose a new scheduler called D-LATE (data-aware LATE). The algorithm of D-LATE can be stated as shown in Algorithm 2. We denote the number of speculative tasks that can be running at once as SpeculativeCap. The progress rate of a task is compared against SlowTaskThreshold, and then it is determined whether the rate is "slow enough" to launch the task speculatively. Our D-LATE algorithm is almost the same as LATE, except that a task is assigned as nearly as possible to the location where the needed data are present. We have implemented an optimized task scheduler based on the D-LATE algorithm in HPMR.

Algorithm 2 D-LATE algorithm

```
INPUT: $T_{free}$ /* the set of free task slots */,
$N_{spec}$ /* the number of speculative tasks */,
$T_{run}$ /* the set of running tasks */
if $T_{free}$ != $\emptyset$ and $|T_{spec}|$ < SpeculativeCap then
    $T \leftarrow T_{run}$;
    while ($T$ != $\emptyset$) do
        Find the task $t \in T$ which has the largest EST
        (estimated time left);
        if The progress rate of $t$ < SlowTaskThreshold then
            Perform speculative execution for the task $t$;
            /* Reassign the task to the node that is the fastest
            and the nearest to the needed data */
            break;
        end if
        $T \leftarrow T$ - $\{t\}$;
    end while
end if
```

IV. IMPLEMENTATION

This section explains the architecture and implementation details of HPMR. As shown in Figure 5, HPMR consists of three main modules: *the optimized scheduler*, *the prefetcher* and *the load balancer*. Currently, these modules are integrated with Hadoop-0.18.3 via the HPMR plug-in interface. This plug-in interface is so flexible that HPMR can support any version of Hadoop or another similar platforms such as GFS [2] and Dryad [9]. In particular, three main modules can be selectively plugged in or out depending on the plug-in interface configuration.

A. Optimized Scheduler

The optimized scheduler is a flexible task scheduler with the predictor and the D-LATE module. In order to understand the optimized scheduler, one must first understand the native Hadoop scheduler. The native Hadoop scheduler assigns tasks to available task trackers by checking heartbeat messages from task trackers. Every task tracker periodically sends a heartbeat message to the job tracker. The job tracker has a task scheduler which actually schedules tasks.

Figure 6 shows how the native Hadoop scheduler works. First, it constructs a list known as *TaskInProgress List*, a collection of all running tasks, and caches the list into either *nonRunningMap* (for map tasks) or *nonRunningReduce* (for reduce tasks) depending on the type of each task. These cached lists are used for the job tracker to manage current map tasks or reduce tasks to be executed. Next, task trackers periodically send a heartbeat message to the job tracker using heartbeat message protocol. Finally, the scheduler assigns each task to a node randomly via the same heartbeat message protocol. The best case of scheduling a task is when the scheduler locates the corresponding task into the local node. The second best case is when the scheduler locates the task into the local rack. In our shared cluster environment, however, the best case and the second best case account for less than 2% and 10% of the total task scheduling, respectively.

The native Hadoop scheduler is too simple to schedule tasks flexibly in the shared MapReduce environment. Especially, the algorithm for predicting stragglers does not work well in the native Hadoop because it uses a single heuristic variable for detection. Under the shared environment, such algorithm mispredicts stragglers, thus randomly assigns tasks.

For these reasons, HPMR provides the optimized scheduler which has the predictor module. The scheduler not only finds stragglers, but also predicts candidate data blocks (during the prefetching stage) and the reducers into which the key-value pairs are partitioned (during the pre-shuffling stage). The information on the expected data is sent to the corresponding task

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on December 11, 2009 at 10:32 from IEEE Xplore. Restrictions apply.

TABLE I
TEST SETS USED IN EXPERIMENTS

	1	2	3	4	5	6	7
Workload	wordcount	wordcount	wordcount	wordcount	wordcount	aggregator	similarity calculator
# nodes	5	10	20	200	200	200	200
# map tasks	18	18	18	50	36	36	36
# reduce tasks	1	1	1	1	1	1	1
Input file size	4.4GB	4.4GB	4.4GB	4.4GB	4.4GB	4.4GB	4.4GB
Input split size	128MB	128MB	128MB	128MB	128MB	128MB	128MB

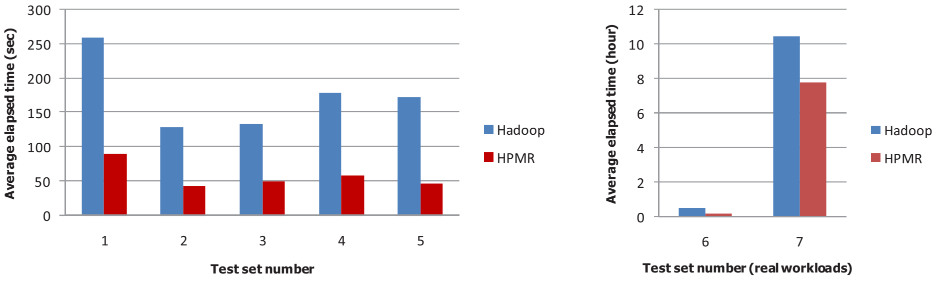


Fig. 7. A comparison of the average elapsed time for each workload

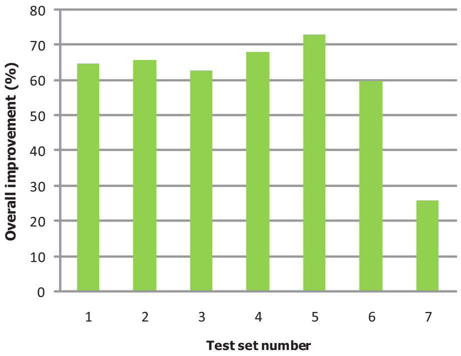


Fig. 8. The overall improvement of HPMR compared to the native Hadoop

queue. In addition to these predictions, the optimized scheduler performs the D-LATE algorithm presented in Algorithm 2.

B. *Prefetcher*

The prefetcher module consists of a single prefetching manager and multiple worker threads. The role of the prefetching manager is to monitor the status of worker threads and to manage the prefetching synchronization with processing bars as mentioned in Section III-A.

Since Hadoop provides several interfaces for HDFS, we have extended two of these interfaces, FSInputStream and FSDataInputStream, and integrated them into a single HPMR plug-in interface. Each worker thread implements this HPMR interface to prefetch predicted data from HDFS. Upon a request from the scheduler, the prefetching manager is invoked, and then it forces worker threads to start prefetching.

C. *Load Balancer*

In HPMR, the load balancer is used when the prefetcher is working for the inter-block prefetching. By contacting the job tracker through the HPMR interface, the load balancer periodically checks the logs, which include disk usage per node and current network traffic per data block. After the inter-block prefetching, the load balancer is invoked to maintain load balancing based on disk usage and network traffic (cf. Section III-A).

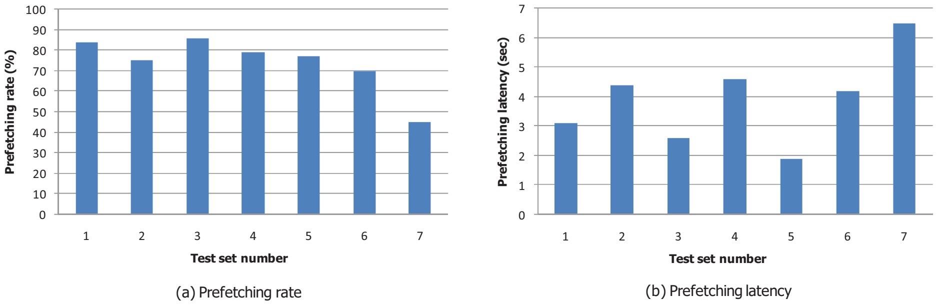
When disk access ratio and network access usage go over their threshold values, the load balancer replicates corresponding heavy-traffic data block into an available low-traffic node in a pipelined manner. After the replication has been completed, our load balancer indicates the task, which have used heavy-traffic data block, as a straggler, and then the task scheduler automatically reassigns the task. From this point on, this task can use the new replicated block in the low-traffic node. As a result, the load balancer effectively manages overall traffic overhead. Our load balancer is executed only when it is invoked by the inter-block prefetching, to avoid wasting system resources caused by frequent load balancing jobs.

V. EVALUATIONS

A. *Experimental Environment on Yahoo!Grid*

The evaluation of HPMR has been performed on Yahoo!Grid, which consists of 1670 nodes. Each node is equipped with two dual-core 2.0GHz AMD processors, 4GB of main memory, four 400GB ATA hard disk drives, and a Gigabit Ethernet network interface card. The entire nodes are divided into 40 racks which are connected with L3 routers. Currently, the number of maximum nodes that can be allocated to each Yahoo! engineer is limited to 200 nodes, and the average number of simultaneous users is approximately 50 users. In our experiments, we vary the number of nodes from 5 to 200. In all tests, we have configured that HDFS maintains four replicas for each data block, whose size is 128 MB. We have performed our evaluations with three different

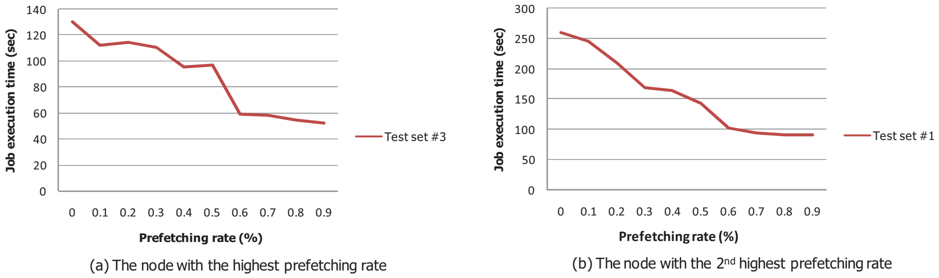
Authorized licensed use limited to: Sungkyunkwan University. Downloaded on December 11, 2009 at 10:32 from IEEE Xplore. Restrictions apply.



(a) Prefetching rate

(b) Prefetching latency

Fig. 9. The prefetching rate and the prefetching latency in HPMR



(a) The node with the highest prefetching rate

(b) The node with the 2^{nd} highest prefetching rate

Fig. 10. Job execution time vs. prefetching rate in test set #3 and test set #1

workloads and seven types of test sets. The first workload is the wordcount, one of the main benchmarks used for evaluating the performance of Hadoop at Yahoo! To emulate complex job of wordcount, we slightly modified the original wordcount to have 1ms delay with the *sleep* function whenever counting words.

The second workload is the search log aggregator, a real MapReduce program used to aggregate monthly search logs and to vectorize terms at Yahoo! The third workload is the page similarity calculator, also a real MapReduce program used to calculate similarity between pages and to apply the data mining technique of the Euclidean function at Yahoo! On each page, it performs a matrix computation, so its complexity is always $O(P^2)$ where P denotes the number of pages. Table I summarizes the characteristics of test sets used in this paper. For each test set configuration, we compare the performance of HPMR with that of native Hadoop in the following subsections.

B. Execution time of overall job

We first compare the overall performance between HPMR and the native Hadoop. For the overall performance, we have measured the elapsed time to complete the corresponding workload in our platform. Figure 7 illustrates the average elapsed time for HPMR and the native Hadoop. We can observe that HPMR shows significantly better performance than the native Hadoop for all of test sets. Figure 8 summarizes the overall improvement of HPMR compared to the native Hadoop. In brief, we can see that HPMR reduces the average execution time by up to 73%.

Note that the test set #1 shows the worst performance. One of the reasons is that the test set #1 has the smallest ratio of the number of nodes to the number of map tasks. This will increase *the scheduling overhead* or the time needed to assign tasks during the execution of the test set #1. The second reason is that it is difficult to exploit data locality when assigning tasks because the least number of nodes is used in the test set #1. The necessary data blocks are more likely to be located at other nodes. Therefore, although the test sets #1 and #2 use the same input file, the performance of test set #1 shows the worse performance than the test set #2.

We can see that the test set #5 exhibits the highest improvement among all the workloads. This is mainly due to significant reduction in the shuffling overhead. Although the test sets #4 and #5 have the same number of map nodes, the test set #4 has much larger number of map tasks. This means that the shuffling overhead of the test set #4 is larger than that of the test set #5. Hence, the average job execution time of the test set #4 will be significantly prolonged due to the shuffling overhead.

On the contrary, the test set #7 shows the smallest improvement over the native Hadoop. The reason is that HPMR fails to predict some candidate blocks for the matrix computation, resulting in poor prefetching rate. We analyze the prefetching rate in each workload in more detail in the next subsection.

C. Effective prefetching rate

We have analyzed the average prefetching rate in each workload by measuring the total size of successfully prefetched data. We have also obtained the average prefetching latency by measuring the time spent on prefetching a block. Although prefetching is performed simultaneously with computation, the prefetching latency is affected by disk overhead or network

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on December 11, 2009 at 10:32 from IEEE Xplore. Restrictions apply.

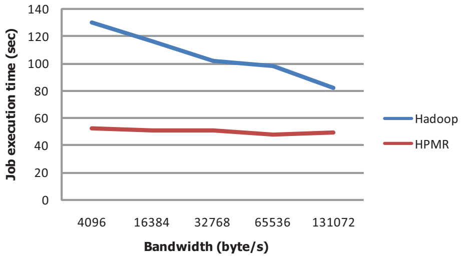


Fig. 11. The effect of bandwidth variation (test set #3)

congestion. Therefore, the long prefetching latency indicates that the corresponding node is heavily loaded.

Figure 9 compares the prefetching rate and the prefetching latency for each test set. Comparing Figure 9(a) with Figure 9(b), it is apparent that the prefetching rate is closely related to the prefetching latency, being inversely proportional to the prefetching latency. Figure 9(a) indicates that the prefetching rate is higher than 70% in almost every workload except for the test set #7. In the test set #7, HPMR was unable to predict some candidate blocks to prefetch due to the random matrix computation needed for similarity calculation.

Figure 10 plots the relationship between the job execution time and the prefetching rate. In this experiment, we only exhibit the results of two representative test sets, #3 and #1, which show two highest prefetching rates among all the test sets. From Figure 10, we can see that the job execution time is saturated at the prefetching rate of 60%. This means that the overall performance remains essentially the same as the prefetching rate increases beyond 60%.

D. Execution time with bandwidth adjusting

Finally, in order to investigate the impact of the bandwidth on the overall performance, we have measured the job execution time varying the available bandwidth manually from 4KB/sec to 128KB/sec. Since all the test sets show the similar trend, we display only the result of the test set #3 in Figure 11. Figure 11 depicts the average job execution time according to the bandwidth variation. In Figure 11, we can observe that HPMR provides almost the same performance independent of the available bandwidth. This means that HPMR assures consistent performance even in the shared environment such as Yahoo!Grid where the available bandwidth fluctuates severely.

VI. Conclusion

In this paper, we propose two innovative schemes, the prefetching scheme and the pre-shuffling scheme, that can improve the overall performance effectively in the shared MapReduce computation environment. The prefetching scheme exploits data locality, while the pre-shuffling scheme significantly reduces the network overhead required to shuffle key-value pairs. The proposed schemes are implemented in HPMR, as a plug-in type component for Hadoop. Through extensive evaluations with five synthetic workloads and two real workloads, we have demonstrated that HPMR improves the overall performance by up to 73% compared to the native Hadoop.

As the next step, we plan to evaluate a more complicated workload such as HAMA [5]. HAMA is a parallel matrix computation package based on the MapReduce programming model which supports linear algebra, computational fluid dynamics, statistics, graphic rendering, and many more. Although HAMA is in very early stage of development, it is currently available as an official open-source Apache incubator project, in which the authors are actively participating.

Acknowledgment

We would like to thank anonymous reviewers and Yahoo! Search Engineering team for their support and collaboration. Especially Heewon Jeon and Seung Park gave us impressive intuitions to initiate this work.

This work was supported by the IT R&D program of MKE/KEIT. [2009-F-039-01, Development of Technology Base for Trustworthy Computing]

References

[1] J. Dean and S. Ghemawat. MapReduce:Simplified Data Processing on Large Clusters. In the Proceedings of the 6th Symposium on Operating Systems Design and Implementation, Dec. 2004.
[2] S. Ghemawat, H. Gobioff, et al. The Google file system. In the Proceedings of 19th Symposium on Operating Systems Principles, Oct. 2003.
[3] Hadoop: http://hadoop.apache.org/
[4] Hadoop On Demand: http://hadoop.apache.org/core/docs/r0.18.3/hod.html
[5] HAMA: http://incubator.apache.org/hama/
[6] E. Nightingale, P. Chen, et al. Speculative execution in a distributed file system. ACM Transactions on Computer Systems, Nov. 2006.
[7] C. Olston, B. Reed, et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In the Proceedings of the ACM SIGMOD international conference on Management of data, June 2008.
[8] M. Zaharia, A. Konwinski, et al. Improving MapReduce Performance in Heterogeneous Environments. In the Proceedings of the 8th Symposium on Operating Systems Design and Implementation, Dec. 2008.
[9] M. Isard, M. Budiu, et al. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Mar. 2007.
[10] R. Lammel. Googles MapReduce Programming Model. Revisited. Science of Computer Programming, July 2007.
[11] H. Yang, A. Dasdan, et al. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In the Proceedings of the ACM SIGMOD international conference on Management of data, June 2007.
[12] M. Burrows, The Chubby lock service for loosely-coupled distributed systems. In the Proceedings of In the Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Nov. 2006.
[13] G. Blelloch. Programming parallel algorithms. In Communications of the ACM, Mar. 1996.
[14] A. Fox, S. D. Gribble, et al. Cluster-based scalable network services. In ACM SIGOPS Operating Systems Review, Dec. 1997.
[15] J. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In the Proceedings of the USENIX Annual Technical Conference, June 2002.
[16] V. Padmanabhan and J. Mogul. Using predictive prefetching to improve world wide web latency. In Proceedings of the ACM SIGCOMM Conference, July 1996.
[17] J. Vitter and P. Krishnan. Optimal prefetching via data compression. Journal of the ACM, Sep. 1996.
[18] T. Kroeger and D. Long. Design and Implementation of a Predictive File Prefetching Algorithm. In the Proceedings of the USENIX Annual Technical Conference. June 2001.
[19] P. Cao, E. Felten, et al. A study of integrated prefetching and caching strategies. In the Proceedings of the SIGMETRICS Conference, May 1995.

Authorized licensed use limited to: Sungkyunkwan University. Downloaded on December 11, 2009 at 10:32 from IEEE Xplore. Restrictions apply.