Scheduling Algorithm based on Prefetching in MapReduce Clusters

Mingming Sun[a,*], Hang Zhuang[a], Xuehai Zhou[a], Kun Lu[a], Changlong Li[a]

[a] *Computer Science University of Science and Technology of China Hefei, China*

Abstract

Due to cluster resource competition and task scheduling policy, some map tasks are assigned to nodes without input data, which causes significant data access delay. Data locality is becoming one of the most critical factors to affect performance of MapReduce clusters. As machines in MapReduce clusters have large memory capacities, which are often underutilized, in-memory prefetching input data is an effective way to improve data locality. However, it is still posing serious challenges to cluster designers on what and when to prefetch. To effectively use prefetching, we have built HPSO (High Performance Scheduling Optimizer), a prefetching service based task scheduler to improve data locality for MapReduce jobs. The basic idea is to predict the most appropriate nodes for future map tasks based on current pending tasks and then preload the needed data to memory without any delaying on launching new tasks. To this end, we have implemented HPSO in Hadoop-1.1.2. The experiment results have shown that the method can reduce the map tasks causing remote data delay, and improves the performance of Hadoop clusters.

Keywords: data locality, MapReduce, prefetching, task scheduler, memory, big data

*Corresponding author at: Computer Science University of Science and Technology of China. Tel: +86 15862451676

Email addresses: mmsun@mail.ustc.edu.cn (Mingming Sun), zhuangh@mail.ustc.edu.cn (Hang Zhuang), xhzhou@ustc.edu.cn (Xuehai Zhou), local@mail.ustc.edu.cn (Kun Lu), liclong@mail.ustc.edu.cn (Changlong Li)

1. Introduction

MapReduce [8] has been highly successful as a parallel distributed processing framework in implementing big data applications on commodity cloud computing plateforms such as Amazon EC2 and Windows Azure. MapReduce divides a computation into multiple small tasks and lets them run on different machines in parallel. It enables hiding the details of the underlying parallel processing to provide a simple programming interface for developing distributed application. There are many different implementations of MapReduce framework such as Hadoop[20], Disco [14], Phoenix [22], etc.

In most state-of-the-art cloud cluster systems, a key challenge is to increase the utilization of MapReduce clusters. If map tasks are scheduled to nodes without input data, these tasks will issue remote I/O operations to copy the data to local nodes. This data transfer delay is primarily on the execution time cost of map phase, while map phase often dominates the execution time of the MapReduce jobs. So data locality becomes one critical factor to affect performance of MapReduce framework. In practice, not only clusters are shared by multiple users, 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 map tasks. The process will cause remote data access delay, thus degrading the performance of MapReduce. Workloads from Facebook and Microsoft Bing datacenters show that this remote I/O operations phase constitutes 79% of a job's duration and consumes 69% of the resources [3]. So in this paper, we focus on the optimizing the map phase. Obviously, the performance of MapReduce clusters is closely tied to its task scheduler. Zaharia [23][24] proposed a delay scheduling algorithm to reduce the map tasks executing remote I/O operations. A next-k-node scheduling method is proposed to improve the data locality [28]. However, in both methods task fairness withered as the cost.

Data prefetching [5] is a data access latency hiding technique, which decouples and overlaps data transfers and computation. And machines in MapReduce clusters have large memory capacities, which are often underutilized; the median and 95 percentile memory utilizations in the Facebook cluster are 10% and 42%, respectively [3]. In light of this trend, we investigate memory locality to speed-up MapReduce jobs by prefetching and caching their input data. Seo et al. [16] designed a intra-block and inter-block prefetching scheme to improve data locality of map tasks, which are assigned to nodes without input data. A data prefetching mechanism in het-

erogeneous or shared environments [9] is proposed. However, both techniques not only cannot reduce the occurrence of such map tasks, but also do not consider the remote access delay of the first data block split.

The prefetching accuracy is the key factor that affects performance. In MapReduce clusters, task scheduler determines the mapping between tasks and nodes. To this end, we design HPSO, a prefetching service based task scheduler to improve performance for MapReduce clusters. The method first predicts the execution time of map tasks and further evaluates the sequence that nodes free busy slots. According to this node sequence, HPSO predicts and assigns the most suitable map tasks to nodes ahead of time. Once such scheduling decisions are made, nodes preload the related input data from remote nodes or local disk to memory before tasks is launching. In this way, input data prefetching is carried out concurrently with data processing, thus data transfer overhead is overlapped with data processing in the time dimension. In summary, in this paper we claim following contributions:

- We provide a novel prefetching mechanism to coordinately manage prefetching input data blocks.
- We exploit task scheduler to determine what and when to prefetch. This method can greatly improve the efficiency of map tasks.
- We have built the High Performance Scheduling Service(HPSO), which combines the task scheduler, prediction, and prefetching mechanism together to exploit data locality and reduce network overhead. To evaluate its performance, we have built a Hadoop cluster system. Compared with the native Hadoop, our implementation of HPSO has demonstrated performance.

The remainder of the paper is structures as follows. Section 2 introduces some technology background necessary to understand the MapReduce and scheduler, and motivation. Section 3 describes the prefetching technique. The design and implementation of HPSO are illustrated in Section 4. Section 5 evaluates HPSO. Related work is described in Section 6. Finally, conclusion is given in Section 7.

2. Background and Motivation

In this section, we introduce the background of MapReduce programming framework, hadoop scheduler mechanism and data locality problem.

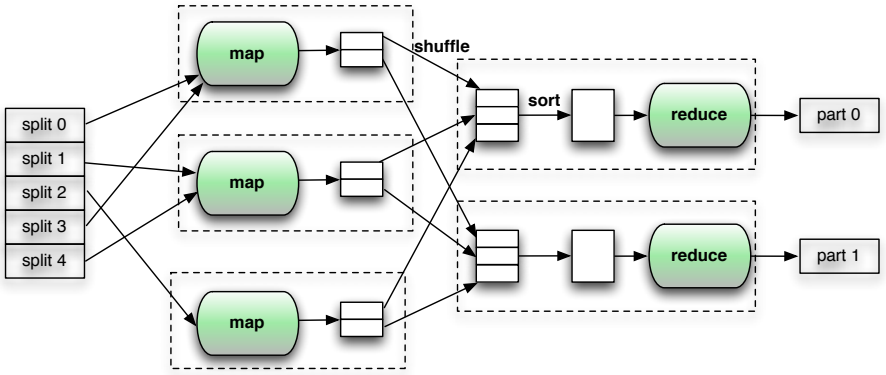
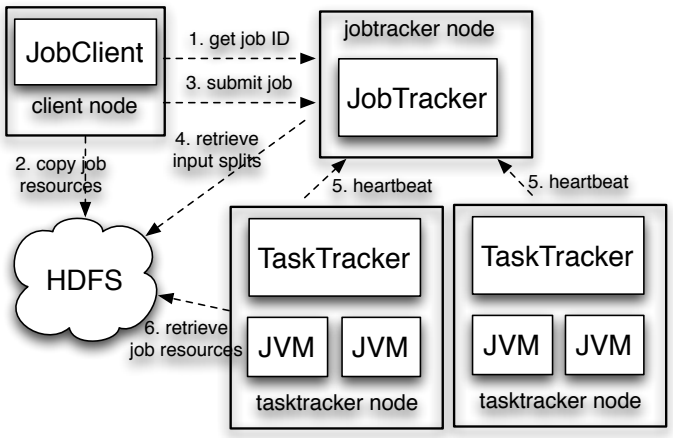


Figure 1: The MapReduce framework.

2.1. MapReduce Programming Framework

Computations in MapReduce framework are divided into map and reduce phases, separated by an internal grouping of the intermediate results as shown as Fig. 1. The power of MapReduce is that the map and reduce functions are executed in parallel over hundreds or thousands of processors with minimal effort by the user. After user submits a job, MapReduce jobs run as follows. Firstly, input data is divided into several fixed-size splits, each of which runs a map task. Then after all map tasks are finished, the intermediate data is reassigned to reduce tasks according to different keys generated in map phase. Reduce phase can be divided into three parts, shuffle, sort and reduce function. The shuffle phase transfers intermediate data generated by map tasks to the corresponding reduce tasks. The sort phase merges all the intermediate data belonging to the reduce task and maintains sort. Then reduce function is called for each key in the sort phase, and directly writes output to distributed file system.

In our paper, we chose Hadoop since it is an open-source implementation of MapReduce model. Furthermore it has been used by many companies such as Yahoo!, Amazon, Facebook duo to its high performance, reliability, and availability. Specially, Hadoop manages computing resources by the term of slot, the basic resource allocation unit. The precise number of slots of each slave in Hadoop cluster depends on the number of cores and the amount of memory. Each slave node provides a number of slots for map tasks and a number of slots for reduce tasks, and these are set independently. Each slot



2.pdf

Figure 2: How Hadoop runs a MapReduce job.

can only run a task simultaneously.

Hadoop Distributed File System (HDFS)[17] is a highly fault-tolerant distributed file system designed to provide high bandwidth for MapReduce by replicating and partitioning files across many nodes. Files in HDFS are automatically partitioned to chunks (64M by default). In MapReduce framework, each map function executes on one chunk.

2.2. Hadoop Scheduler

Fig. 2 illustrates the work mechanism of Hadoop running a MapReduce job. Hadoop clusters have one JobTracker, which coordinates the job run, and a number of TaskTrackers, which are in charge of running jobs and periodically heartbeat the JobTracker. First, when JobTracker receives a client submitted job, it puts it into an internal queue from where the job scheduler will pick it up. Then the job scheduler retrieves the input splits of job, and creates one map task for each split and creates reduce tasks according to mapred.reduce.tasks property set by client. Second, When a TaskTracker indicates that it has an idle map slot by heartbeat, job scheduler will allocate it a map task, otherwise, it will select a reduce task.

A computation requested by a job will be performed much more efficiently if it is executed near the data it operates on. However, as HDFS files are divided across all nodes, some map tasks must read data over the network. One of Hadoop basic principles is moving computation is cheaper than mov-

ing data. Therefore, for a map task, it takes into account the TaskTracker's network location and picks a task whose input data is as close as possible to the TaskTracker. The scheduling policy preferentially selects the tasks with data locality. In the optimal case, map task is data-local, that is, running on the same node that input data resides on. Alternatively, the next best case is when the data is in any other node within the same rack, called rack locality. Some map tasks retrieve their data from a different rack, rack-off locality.

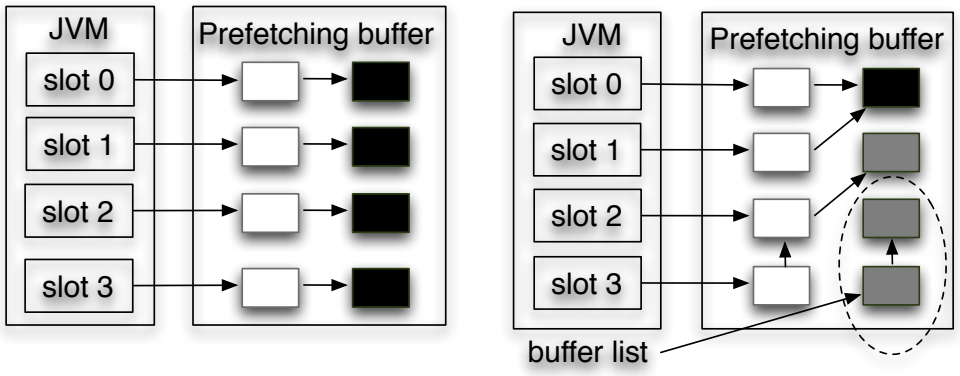
2.3. Motivation

TaskTracker will not process a map task until its input data is loaded into the tasktracker node's memory. Map tasks with rack locality or rack-off locality cause remote data transmission overhead and can slow down the entire job. And the network bisection bandwidth in a large cluster is much lower than the aggregate bandwidth of the disks in the nodes, so data locality issue becomes crucial for performance of MapReduce clusters. Especially, data locality issue suffers in two situations: concurrent jobs and small jobs[23]. Indeed, workloads in MapReduce clusters consist of many small jobs and relatively few large jobs. It is necessary to research a method to hide data transmission overhead.

As machines in Hadoop clusters have large memories, in-memory data prefetching is an efficient way to solve this problem. Prefetching can hide data transmission delay by preloading the expected data ahead of time. The key challenge, however, is how to improve prefetching rate. That is to determine what and when to prefetch. Some approaches use prediction algorithms based on history of data accesses or cache misses [7][10][11][15]. However, in MapReduce clusters, task scheduler determines to assign tasks to tasktracker nodes. So our method researches prefetching mechanism using task scheduler. Another reason motivated this method is that history information of data is not well reflect the future access to data in cloud computing. We design a scheduling policy which predicts the most appropriate tasktracker nodes to assign tasks.

3. Prefetching

In this section, we present data prefetching mechanism in detail. HPSO is to preload the input data before map task is running on tasktracker node. As a result, prefetching can hide data transmission delay and further



(a) (b)

Figure 3: Prefetching buffer structure and management. The tasktracker node has four map slots. The white box represents a processing block by map task. The black box represents a prefetching block for the following map task, and the gray corresponds a processed block by the previous map task.

improve MapReduce performance. The emphasis of this section is not on the syntactical details of HPSO, but on how to simply and effectively manage the memory buffer for prefetching data.

3.1. Prefetching Buffer Management

The process of data prefetching mechanism is shown as following. When a tasktracker node receive a prefetching request from HPSO, the node will load expected data block to a buffer in memory called prefetching buffer. When the corresponding map task arrives, it will process data of prefetching buffer. Obviously, this process is typical producer-consumer model, where data prefetching thread is the producer and corresponding map task is the consumer. The following issues must be addressed in the prefetching mechanism.

One issue is the size of prefetching buffer. Intuitively, bigger prefetching buffer means better performance. But in fact, according to the producer-consumer model, two buffer units for each map slot are enough. In the paper, prefetching buffer unit is the basic data block of HDFS (by default 64MB), instead of part of basic data block. This allows an uncapped number of

replicas in prefetching buffer, which in turn increases the chances of achieving memory locality especially when there are hotspots due to popularity skew [2]. For example, our method is advantageous for iterative applications, in which each iteration can be expressed as a MapReduce job and need input data. And with the development of technology, memory capacity is increasing. Therefore the memory utilization of the two buffers is unnoticeable to affect the overall performance.

Another issue is how to manage prefetching buffer. Fig. 3 illustrates the prefetching buffer structure and management strategy. Each slot of the tasktracker node has at most two buffer units. One is processing block using by the running map task. The other may be prefetching block, which has been preloaded for the next map task, or null. Fig. 3 (a) shows that all slots have preloaded the data blocks for the following map tasks, and maintain two buffer units into a list. This strategy is convenient to manage prefetching buffer and reuse buffer data. Specifically, a particular data block may be used simultaneously by multiple map tasks. For example, in Fig. 3 (b), *slot 0* and *slot 1* share the same prefetching block. Alternatively, data block which is being processed or has processed may be prefetching block for other slot such as *slot 2* and *slot 3* in Fig. 3 (b). Then the remaining buffer units will be linked into the buffer list. When a buffer unit is needed to store new prefetching data, the method get a buffer unit from buffer list using LRU.

4. HPSO Design and Implementation

In this section, we present the HPSO design issues and implementation, and discuss the techniques required to achieve our goal. Our design seeks to minimize total execution time of applications and improve the performance of MapReduce clusters. The basic idea of HPSO is to overlap the data transmission process of the following map task with data processing process of running map task. The emphasis of this section is that how to effectively design prefetching requests based scheduling policy.

4.1. Framework

As shown in Fig. 4, HPSO consists of three main modules: the prediction module, the scheduling optimizer and the prefetching module. The role of scheduling optimizer is to predict the most appropriate tasktracker nodes to which future map tasks should be assigned. Once the scheduling decisions are made before map tasks are scheduled, HPSO will trigger the prefetching

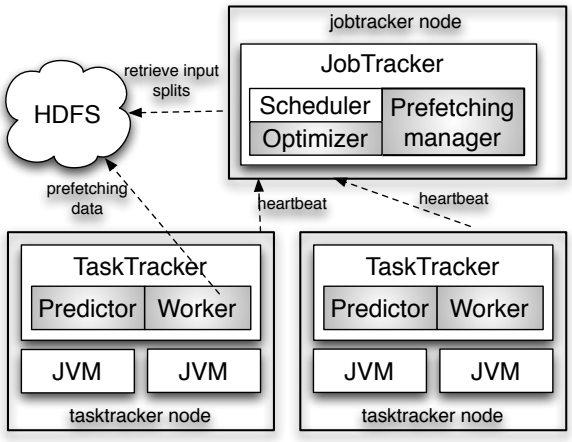


Figure 4: The architecture and framework of HPSO

module to load expected input data. Then our method can explore the underutilized disk bandwidth or network bandwidth in CPU-intensive process. Such pipelining can hide away data transfer latency. To implement pipelining, the prediction module in each tasktracker node predicts the remaining execution time of map tasks and further evaluates the sequence that slot become idle.

The prefetching module adopts a master-slave architecture with one prefetching manager and in JobTracker and a set of workers located at TaskTrackers. Prefetching manager is responsible for monitoring the status of workers and coordinating the prefetching process for map tasks. Each worker can automatically finish loading data block by itself before the map task is received. When prefetching manager achieves prefetching instructions from scheduling optimizer, prefetching manager triggers workers to load data to memory. Prefetching manager also reports scheduling optimizer the data blocks in TaskTrackers' prefetching buffer.

An example of prefetching using HPSO is shown in Fig. 5, where the data block $A1$ required by $map1$ are prefetching from node 2 to node 1 in pipelined manner when $map1$ is predicted to will be assigned to node 1. Thus, HPSO can hide away data transfer latency.

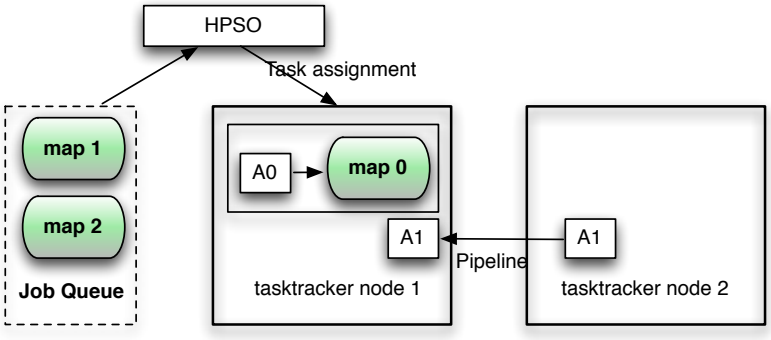


Figure 5: An example of prefetching using HPSO. *map* 1 processes the data block $A1$, which only resides on node 2. HPSO predicts that this task will be assigned to node 1. Node 1 prefetching $A1$ in a pipelined manner.

4.2. Node Prediction

In Hadoop, tasktracker node requests a map task when it frees a busy slot. And the sooner a tasktracker node has an idle slot, the earlier the node requests a new map task. So tasktracker to issue a request can be predicted according to the time of each tasktracker to complete tasks. That it can be measured by the remaining time of map tasks running on the node. Hadoop monitors task process using a process score, which is a number from 0 to 1. For a map, the score is the fraction of input data read. We estimate the remaining time of an executing map task based on current process using Eq. (1), which is proposed in Ref. [26].

$$RT_m = \frac{t_{exe}}{s_{exe}} * (1 - s_{exe}) \tag{1}$$

In the method, RT_m is the time left of map task, and t_{exe} is the execution time of the task when the process is up to s_{exe}. In MapReduce framework, a node normally runs multiple tasks simultaneously. For each slot, we compute Eq. (1) and sort results by ascending order. Then we can get the sequence that slots become idle.

$$T_m = \frac{t_{exe}}{s_{exe}} \tag{2}$$

$$t_{exe} = t_{tran} + t_{cpu} \tag{3}$$

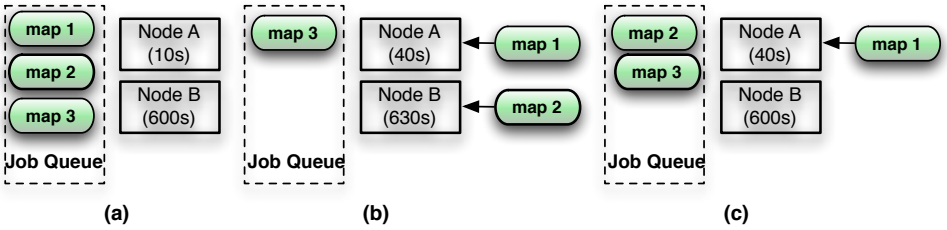


Figure 6: An example of assigning tasks in advance. (a) the initial state: three map tasks in job queue and two tasktracker nodes. Supposing each map task spends 30s and the remaining time of node A is 10s, but node B 600s. (b) each node has a waiting task. (c)map tasks are assigned based on HPSO.

$$T_{cpu} = \frac{t_{cpu}}{s_{exe}} \tag{4}$$

Further we can calculate the completed execution time of map task by Eq. (2), instead of waiting task completion. In the implementation of Hadoop, the input data of map tasks is not transferred all to memory, but a mass of small slices instead. When the processing of the data slice is completed, map tasks will transfer and process another data slice. The data transmission and data processing happens in sequence. So the execution time of map tasks is composed of the data transmission time and the data processing time as shown as Eq. (3). The total data processing time can be estimated by Eq. (4). With the completion of map tasks of same job, we calculate the average of the completed execution time of map tasks as the predicted execution time, assuming that each node processes tasks at a roughly constant speed.

4.3. Scheduling Policy

In Hadoop, task scheduler maintains a complete view of which tasks are running on which tasktracker nodes and job waiting queue. HPSO combines this view with the slot sequence predicted in Section 4.2 to map appropriate tasktracker nodes with future tasks from job waiting queue, and then triggers the prefetching module to preload input data for map tasks without input data.

4.3.1. Prefetching based Scheduling Algorithm

In this paper, we assign future tasks to nodes, whose remaining time is less than a threshold called $Tunder$, instead of all nodes. $Tunder$ is greater than the basic data block transmission time. We use N_p to represent node sequence, each of which has less remaining time than $Tunder$. This method has the advantage of avoiding invalid prefetching when some nodes are running long tasks as shown as Fig. 6. If the node sequence is only considered in Fig. 6 (b), both nodes have a waiting map task and prefetch input data. However, in practice, all map tasks are executed on node A. So this method causes invalid prefetching data for $map2$ on node B. Our method only assigns $map1$ to node A using $Tunder$ and avoid invalid prefetching operations.

If the input data of m is not stored on any node in N_p, the task will copy the input data by remote I/O operations. Therefore, m should be targets. Supposing that the input data of each task has three replicas, this task has data locality in these three tasktracker nodes $n_l(l \in [0, 2])$. The model denoted by Eq. (5) is used to select map tasks with no data locality. $u(n_i, n_j)$ checks whether the nodes identified by n_i and n_j represents the same node illustrated by Eq. (6). If the two nodes represent the same node, $u(n_i, n_j) = 1$, and vice versa.

$$f(m, N_p) = \sum_{i=0}^{3-1} \sum_{j=0}^{p-1} u(n_i, n_j) \tag{5}$$

$$u(i, j) = \begin{cases} 1 & if\ i = j, \\ 0 & otherwise \end{cases} \tag{6}$$

If $f(m, N_p) = 0$, the input data of m is not stored on any of the predicted node sequence. m will cause the remote data access delay. Therefore, m should be target. If $f(m, N_p) > 0$, at most one node stores the input data of m. It is possible that there are more than one task whose input data is stored on same node. To avoid such occurrence, we assign the first node-local map task to it.

Algorithm 1 outlines the basic steps of prefetching based scheduling algorithm. HPSO considers the data blocks in prefetching buffer. If input data block of map task has been in prefetching buffer of certain node, we called this buffer-local and preferentially assign the task to this node. The data in prefetching buffer is equivalent to one replica, which in turn increases the

chances of achieving data locality. Firstly, algorithm scans all highest priority map tasks and all nodes in N to assign potential buffer-local tasks. Then array N will be ascendingly reorder according to $RT_n = RT_n + T_{cpu}$, where RT_n is the remaining time left of n and T_{cpu} is the processing time of task t. Secondly, If array N is not null, our algorithm computes Eq .(5) to fully exploit the potential map tasks with data locality. If a node-local task is found, HPSO will trigger prefetching module to cache data from disk and ascendingly reorder array N. Unfortunately if we must select rack-local or rack-off

Algorithm 1 Prefetching based Scheduling Algorithm

Input: Array N: the predicted slot sequence whose remaining time is less than $Tunder$; Array J: job waiting queue.

Begin:

```
1: if !( All slots in N have at least one waiting task or J ==NULL) then
2:     for j in J do
3:         if j has a buffer-local task t for n(n in N) then
4:             set M < n, t >; Break
5:         end if
6:     end for
7: end if
8: if !( All slots in N have at least one waiting task or J ==NULL) then
9:     for j in J do
10:        compute f(m, N_p) using Equation (5)
11:        if f(m, n) >= 1 then
12:            set M < n, t >;
13:        end if
14:    end for
15: end if
16: while !( All slots in N have at least one waiting task or J ==NULL) do
17:    for j in J do
18:        if j has a rack-local task t for n(n: the head slot of N) then
19:            set M < n, t >; Break
20:        else
21:            set M < n, t >;
22:        end if
23:    end for
24: end while
```

task, HPSO will make prefetching instructions and trigger prefetching module to preload data from remote node. The prefetching instructions contain expected data block list and the corresponding destination tasktracker nodes, and where the required data blocks are located.

In addition, when a new job is first submitted, there is no data available. In this paper, the execution time of new map task of the job is considered infinity. When this map task is assigned in advance to a tasktracker node, the node will be sorted to the tail of node queue that slots become idle. Once the map task starts to execute, the remaining time of map task will be predicted. Thus the node will be reordered.

4.3.2. Scheduling Algorithm

HPSO guarantees all map tasks with data locality to the maximum extent. One important principle is that the method does not affect the priority of jobs. In this paper we suppose that map tasks in the same priority can be executed out of order. However, inaccurate prediction may cause that low priority tasks will be executed earlier than high priority tasks. To address this problem, when JobTracker receives a heartbeat from node n and n has a idle slot, we first determine whether there is a higher priority map task than task t, which is assigned to n in accordance with prefetching based scheduling algorithm. Secondly, map take p with the longest waiting time left is chose. Then task scheduler calculates Eq. (7). RT_m is the waiting time left on node m, where p is assigned in advance. T_{tran} is the data transmission time. If u is positive, we will remove p from original predicted node m and assign it to n. Otherwise, HPSO will assign original predicted map task to it. Formal pseudocode for scheduling algorithm is as shown as Algorithm 2.

To this end, HPSO does not affect the priority of jobs. Another issue is fault tolerance. HPSO's failure does not hamper the job's execution since input data can always be read from remote nodes or local disk.

$$u = RT_m - T_{tran} \tag{7}$$

4.4. Prefetching Module

The prefetching module consists of a central prefetching manager and a set of workers located at tasktracker nodes as shown as Fig. 7. The prefetching manager constructs a list known as the data list for each tasktracker node, a mapping between every prefetching block and tasktracker nodes that cache

Algorithm 2 Scheduling Algorithm

Input: M $< n, t >$: the mapping between the tasktracker node n and map tasks t which have been assigned to n in advance;

Begin: A heartbeat is received from node n

```
1: if n has a free map slot then
2:     Array A = Higher_priority_choose(t)
3:     if A not NULL then
4:         task p = Max_waiting_time(A)
5:         u = RT_m − T_tran(m is the mapping with task p)
6:         if u > 0 then
7:             return m to n
8:         end if
9:     else
10:        return t to n
11:        Ascendingly reorder array N;
12:    end if
13: end if
```

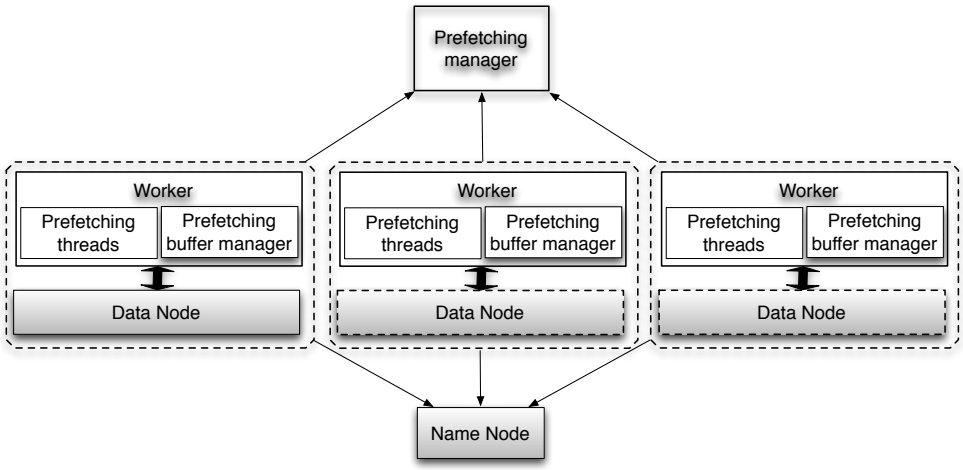


Figure 7: An overview of Prefetching Module.

it. It is worth noting that network bandwidth is one of the critical factors to affect performance of MapReduce clusters. To this end, HPSO combines these data information to make scheduling decisions in order to minimize the

network transmission traffic. For example, it can reduce the cost of loading the released data at the previous round for iterative applications.

Upon the arrival of a prefetching request from HPSO, prefetching manager triggers prefetching threads in tasktracker nodes to start loading corresponding input data to prefetching buffer according to prefetching instructions. Each worker in tasktracker node consists of prefetching threads and prefetching buffer manager. The prefetching thread's main role is to serve cached blocks, as well as prefetch new data blocks. The prefetching thread periodically informs the prefetching manager of data block updates as the part of heartbeat message. The prefetching manager uses these updates to maintain data lists. The role of prefetching buffer manager is to individually manage prefetching buffer illustrated in Section 3.

5. Evaluation

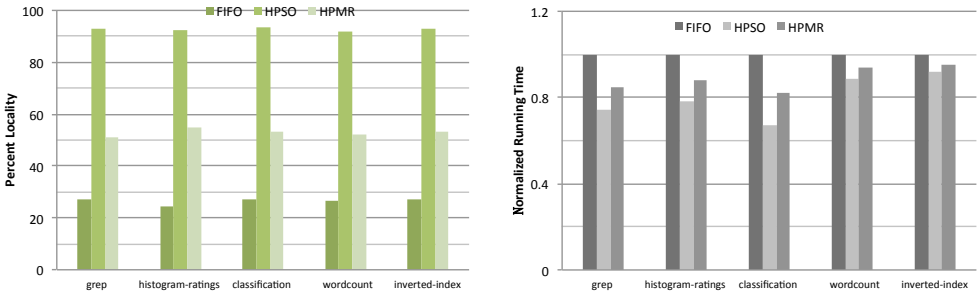
We are going to evaluate HPSO in term of performance and scaling. The performance metric is measured as the improvement over default Hadoop. The scaling metric is measured as a different number of cluster nodes. We first describe our evaluation setup before presenting our results.

5.1. Setup

Platform. To measure HPSO's performance, we have built a Hadoop cluster, which has one master and 20 machines with 16GB of memory, 4 cores and 200GB of storage. A common gigabit Ethernet switch connected each node. The baseline for our deployment is Hadoop 1.1.2 and we configured that HDFS maintains three replicas for each data block in this cluster.

Benchmarks. We performed our evaluations with 5 benchmarks released on PUMA [1], including *grep*, $histogram-ratings$, $classification$, *wordcount* and $inverted-index$. To separate the effort of HPSO on different jobs, we binned them by the number of input data size. In the paper, these benchmarks ran varying numbers of jobs respectively based on the job size so as to take 20-30 minutes in total.

Comparison with Hadoop and HPMR. We compared the execution time of benchmarks and the probability of map tasks with data locality on HPSO with that on both Hadoop and HPMR. We run the benchmarks on HPSO, Hadoop, and HPMR separately and get the average value. Hadoop employed a simple FIFO scheduling policy, which assigns the earliest submitted job to execute, then the second, etc. There is also a priority policy for



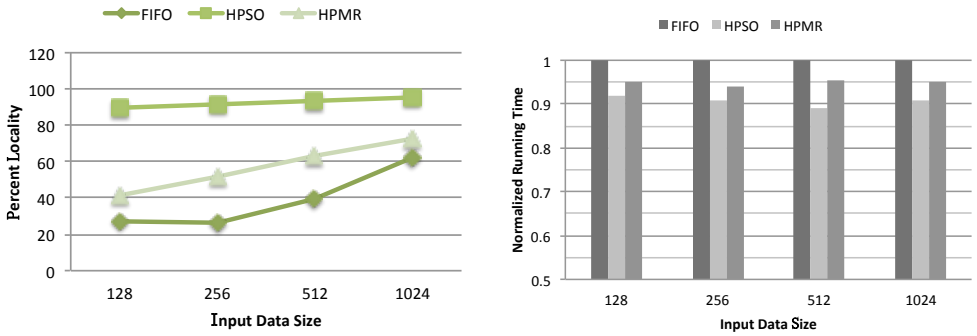
(a) (b)

Figure 8: Comparison for different benchmarks. (a) comparison of the map tasks processed with data locality. (b) comparison of normalized running time.

putting jobs into higher-priority queues. HPMR [16] provides a intra-block and inter-block prefetching scheme to improve data locality of map tasks. The intra-block prefetching technique prefetches a data split within a single block while performing a complex computation. The inter-block prefetching technique prefetches next expected data block to a local rack when the map task simultaneously processes multiple data blocks.

5.2. Performance of HPSO on Benchmarks

Firstly, we designed these tests to evaluate HPSO's performance. These tests run on our Hadoop cluster, which has one master and 15 machines. The entire machines are divided into 3 racks which are connected with L3 routers. And every machine is limited to run at most four map tasks and four reduce tasks simultaneously. Fig. 8 shows the normalized execution time and data locality achieved by FIFO, HPMR and HPSO when running multiple the same jobs. Each job has 256MB input data size. Because the default data block of HDFS is 64MB, each job has 4 map tasks and 1 reduce tasks. We can observe that our method shows significantly higher data locality than FIFO and HPMR for all of test sets in Fig. 8 (a). HPSO raised data locality to at most 93.5% for $classification$ benchmark, and at least 92.1% for *wordcount*. HPMR gets lower probability of map tasks with data locality. Because the intra-block prefetching technique of HPMR only prefetches next input data split within a single block, but does not consider



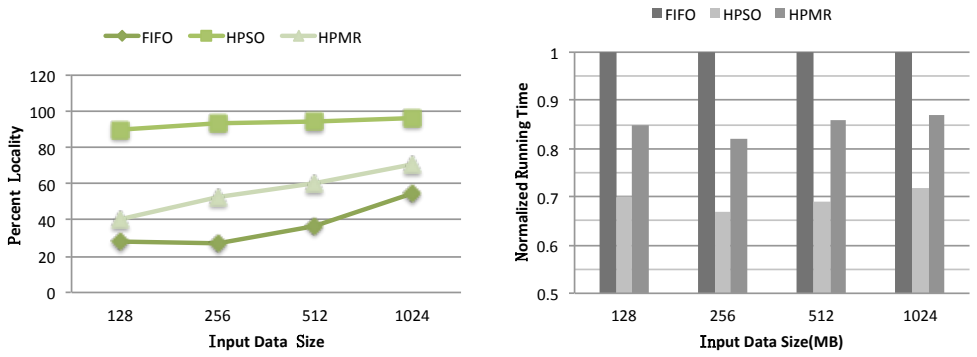
(a) (b)

Figure 9: Comparison of performance for wordcount jobs. (a) comparison of the map tasks processed with data locality. (b) comparison of normalized running time. The horizontal axis shows the different input data size.

the remote access overload of the first input data split. The inter-block prefetching technique prefetches next input data block for map task required multiple data blocks, but does not cosider the remote access overload of the first data block. Correspondingly, HPSO obtains higher performance improvement than HPMR. HPSO finished jobs 33% faster than FIFO and 15% faster than HPMR. In addition, for three jobs *grep*, $histogram-ratings$, and $classification$, the performance improvement is larger than other two jobs in Fig. 8 (b). Because most of time is spending on data IO, prefetching the input data of map accelerates the execution of Map phase significantly and gets speedup of at most 1.49 times. For *wordcount* and $inverted-index$, the shuffle and reduce phases account for a large proportion causing the performance gains to 1.12 time and 1.08 time. Virtually all the gains are due to preloading the input data for rack-local or rackoff tasks. This would increase throughput in a more bandwidth-constrained environment.

5.3. Performance on Different Input Data Size of Jobs

In these tests, we compared the performance of HPSO with that of default Hadoop as different number of input data size. We choose$classification$ and *wordcount* benchmarks. Fig. 9 (b) and Fig. 10 (b) show the normalized running time of two benchmarks, while Fig. 9 (a) and Fig. 10 (a) show locality achieved by FIFO, HPMR and HPSO. In these tests, the input sizes



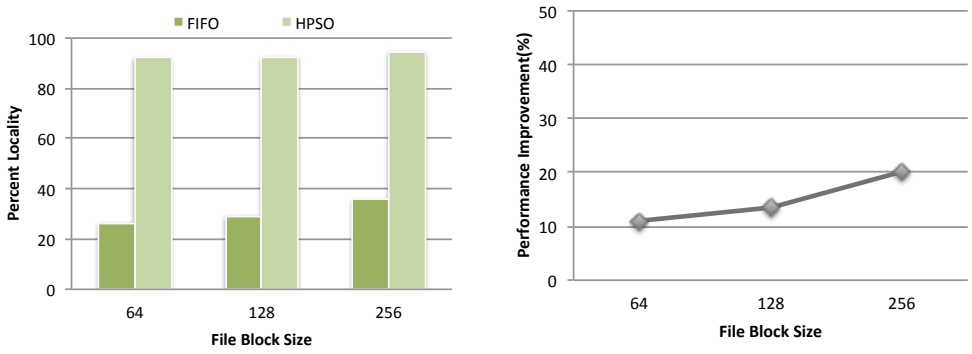
(a) (b)

Figure 10: Comparison of performance for classification jobs. (a) comparison of the map tasks processed with data locality. (b) comparison of normalized running time.

of jobs are 128MB, 256MB, 512MB, and 1024MB, respectively. Correspondingly, jobs have 2 map tasks, 4 map tasks, 8 map tasks and 16 map tasks. HPSO can achieve significantly higher probability of map tasks with data locality than FIFO and HPMR for all of test sets. HPSO raised data locality to at least 89.7% and at most 95.6% for *wordcount*, and at least 90% and at most 96% for $classification$. HPSO increased throughout by at most 11% for 256MB input data and at least 8% for 1024MB input data on *wordcount* with FIFO. Our method outperforms FIFO at most 1.49 times on $classification$ of 256MB input data and at least 1.39 times on 1024MB input data. HPSO improve performance by up to 17.1% with HPMR. The throughput gain is lower at jobs of 1024MB input data than other jobs because locality is fairly good even without data prefetching. However, the gain for the smallest input size jobs is lower than for 256MB and 512MB input size jobs, because at small job sizes, job initialization becomes a bottleneck in Hadoop.

5.4. Performance on Different Data Block Size of HDFS

Fig. 11 (b) shows the performance improvement when the data block size of HDFS is set with different values, from 64MB to 256MB. HPSO has increasingly improved the performance greatly as the data block size becomes larger, from 11% to 20.1%. That is because the data transmission time becomes longer with block size increasingly. Our method improves the



(a) (b)

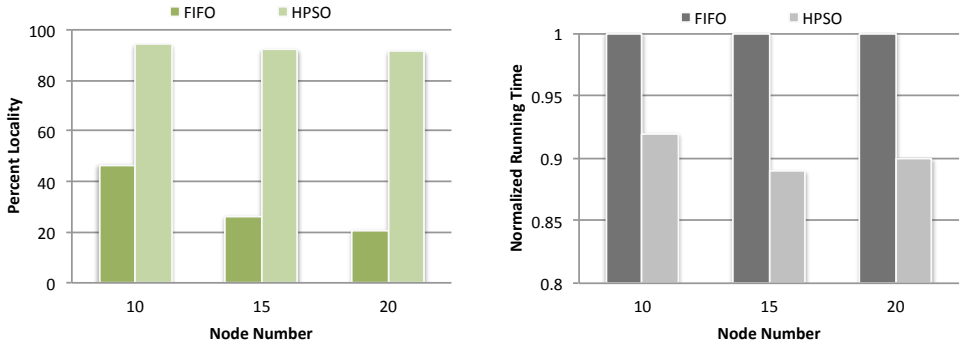
Figure 11: Comparison for different block size for wordcount jobs respectively. (a) comparison of the map tasks processed with data locality. (b) performance improvement of HPSO compared to default Hadoop. The horizontal axis shows the data block size in HDFS.

percentage of map tasks with data locality by prefetching as shown as Fig. 11 (a). We can observe that our method is not substantially affected by the size of the data block. In summary, HPSO raises the percentage of map tasks with data locality by prefetching. Therefore it improves the performance compared with default Hadoop.

5.5. Scaling Performance

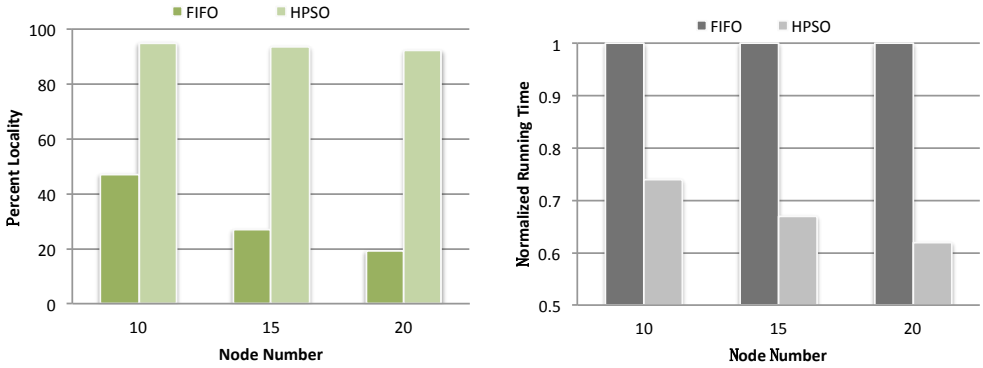
We explored the scalability of HPSO, growing with the number of slave nodes from 10 to 20. In our tests, we also choose$classification$ and *wordcount* benchmarks. Fig. 12 and Fig. 13 suggests that HPSO outperforms FIFO with different nodes in different benchmarks. HPSO reduces the map tasks without data locality of *wordcount* by 5.9% on cluster of 10 slave nodes, and by 7.9% on 15 slave nodes and by 8.5% on 20 slave nodes. Furthermore, HPSO improve performance by prefetching input data. The improvement in data locality for 10 nodes is lowest because locality in this smallest cluster is fairly good. The performance gain in largest cluster is higher than other jobs since the improvement in data locality is the most.

Of course, the only way to conclusively evaluate HPSO's performance at scale will be to deploy it on a large cluster. But in light of this trend, our experiments suggest that HPSO will continue to perform well at scale.



(a) (b)

Figure 12: Comparison for different tasktracker node number for wordcount jobs, respectively. (a) comparison of the map tasks processed with data locality. (b) comparison of normalized running time. The horizontal axis shows the number of tasktracker nodes.



(a) (b)

Figure 13: Comparison for different tasktracker node number for classification jobs, respectively. (a) comparison of the map tasks processed with data locality. (b) comparison of normalized running time.

5.6. Discussion

In general, in terms of performance, HPSO outperforms FIFO and HPM-R. The performance gain is mainly based on hiding the remote input data access delay for map tasks. In terms of scaling performance, results suggest

that HPSO will continue to perform well as cluster size increases to tens of thousands of cores.

In the implementation of HPSO, it needs to occupy part of memory as prefetching buffer. But there is little effect on clusters' performance since memory is often underutilized in clusters [3]. Furthermore, two basic data block units for each map slot are enough, and HPSO occupies little memory for each node. Unfortunately, prediction error may lead to ineffective prefetch, and then causes remote network transmission cost. However, data prefetching in HPSO is performed when memory and network are relatively idle. In detail, we use cache hit rate to measure the memory available. So ineffective prefetch has little effect on the performance. Furthermore, process score is approximately equal to the percent completion of map task, and then can be successfully used to estimate time left of map tasks [26]. Our test results also show that HPSO achieves high data locality. HPSO achieves higher benefits than overhead. Finally, in all tests, our method can raise data locality to at least 89.7%, which is fairly high and acceptable for most state-of-the-art literatures.

6. Related Work

Recently, prefetching and scheduling technology has been used to solve the data locality problem of MapReduce. Zaharia [23][24] presented a delay scheduling algorithm, which addresses the conflict between locality and fairness in shared MapReduce clusters. A next-k-node scheduling method [28] is similar to delay scheduling algorithm, and considers k candidate nodes for each tasks. However both algorithms do not consider other map tasks without data locality and task fairness withered as the cost. Our work optimizes all map tasks.

Considerable work has been carried out on prefetching methods to reduce I/O latency. An affinity- based metadata-prefetching (AMP) scheme [11] is proposed for metadata servers in large-scale distributed storage systems to provide aggressive metadata prefetching. Through mining useful information from historical accesses, AMP can discover metadata file affinities accurately and intelligently. However, this algorithm does not apply in cloud computing environment since its aggressive prefetching. Yong [7] proposed an Algorithm-level Feedback-controlled Adaptive (AFA) data prefetcher to address data-access delay in High-Performance computing by analyzing the data-access history cache. A real-time data prefetching algorithm [10] is pro-

posed based on sequential pattern mining and adopts predictive prefetching technology predict related data objects of data object on demand. Both algorithms focus on analyzing the historical data access records and require to predict uses' behavior. Performance improvement has relationship with the uses' behavior. Our method combines prefetching with task scheduler and prefetches input data of the next running map task ahead of time to hide the data transmission delay. Seo et al. [16] designs a prefetching scheme and a pre-shuffling scheme. However, it cannot reduce the total number of the map tasks without node locality, and the method occupies much network bandwidth, so system performance may be decreased. Compared with the Seo's method, our method can hide the remote access delay of the first data block and handle all map tasks without data locality. A data prefetching mechanism [9] in heterogeneous or shared environments is proposed, but the method also does not consider the first data block transfer delay. The method only deal with intra-block prefetching for map tasks. A prefetching stratrgy [6] only focuses on rack-off map tasks and prefetches input data from different rack. So it improves the performance lower than our method. A predictive scheduler and prefetching mechanism [21] are proposed to improve the performance of MapReduce by assigning two tasks to each slot. Unfortunately, it affects the priority of jobs. Our prior work [18] is also a prefetching based scheduling algorithm to hide the remote data access delay. However, this paper explores the potential tasks with data locality, further reduces the number of data prefetching.

Caching technology also has been used to improve MapReduce performance. PACMan [3] and HaLoop [4] cached input data in memory to reduce IO cost of hard disks and optimize performance. Ref. [25][13] proposed a distributed high-performance storage in memory. mpCache [19] is a SSD based hybrid storage system that caches both input data and localized data. unbinding-Hadoop [12] unbinds map tasks and their input data when using caching and prefetching technology to improve the hit ratio of iterative MapReduce programs. But these methods need more memory capacity. And cache replacement policy is made based on historical data access records. However, task scheduler decides the task assignments. Thus our method can get a higher probability of map tasks with data locality. Other method such as Zhang et al. [27] provides a new method to improve the performance by using distributed memory cache as a high speed access between map tasks and reduce tasks. Map outputs sent to the distributed memory cache can be gotten by reduce tasks as soon as possible.

7. Conclusion

Achieving memory locality for map task will shorten its completion time, further improve the cluster throughout. This paper presents HPSO, which exploits task scheduler to preload required input data prior to launching tasks to TaskTracker. It hides the waiting period of map tasks with rack and rackoff locality and shortens the completion time for MapReduce jobs. HPSO integrates a prediction module and a prefetching module with scheduling optimizer. A scheduling optimizer is integrated into HPSO to improve prefetching rate. We use 5 benchmarks to demonstrate that our method can outperform default Hadoop, and improve data locality at least 89.7% in all tests. In light of these results, we believe that HPSO can achieve better utilization of node resources and a high system throughout in MapReduce clusters.

Acknowledgment

Our work could not have been implemented without the assistance of many individuals and teams. Especially our work was supported by the National Science Foundation of China under grants No. 61272131 and No. 61202053, China Postdoctoral Science Foundation grant No. BH0110000014, Fundamental Research Funds for the Central Universities No. WK0110000034, and Jiangsu Provincial Natural Science Foundation grant No. SBK201240198.

[1] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. Puma: Purdue mapreduce benchmarks suite. 2012.

[2] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, pages 287–300. ACM, 2011.

[3] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *NSDI*, pages 267–280, 2012.

[4] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, 2010.

[5] Surendra Byna, Yong Chen, and Xian-He Sun. A taxonomy of data prefetching mechanisms. In *Parallel Architectures, Algorithms, and Networks, 2008. I-SPAN 2008. International Symposium on*, pages 19–24. IEEE, 2008.

[6] Gaozhao Chen, Shaochun Wu, Rongrong Gu, Yongquan Xu, Lingyu Xu, Yunwen Ge, and Cuicui Song. Data prefetching for scientific workflow based on hadoop. In *Computer and Information Science 2012*, pages 81–92. Springer, 2012.

[7] Yong Chen, Huaiyu Zhu, and Xian-He Sun. An adaptive data prefetcher for high-performance processors. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 155–164. IEEE, 2010.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] Tao Gu, Chuang Zuo, Qun Liao, Yulu Yang, and Tao Li. Improving mapreduce performance by data prefetching in heterogeneous or shared environments. *International Journal of Grid & Distributed Computing*, 6(5), 2013.

[10] Jiazheng Li and Shaochun Wu. Real-time data prefetching algorithm based on sequential patternmining in cloud environment. In *Industrial Control and Electronics Engineering (ICICEE), 2012 International Conference on*, pages 1044–1048. IEEE, 2012.

[11] Lin Lin, Xuemin Li, Hong Jiang, Yifeng Zhu, and Lei Tian. Amp: An affinity-based metadata prefetching scheme in large-scale distributed storage systems. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 459–466. IEEE, 2008.

[12] Kun Lu, Dong Dai, Xuehai Zhou, Mingming Sun, Changlong Li, and Hang Zhuang. Unbinds data and tasks to improving the hadoop performance. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014 15th IEEE/ACIS International Conference on*, pages 1–6. IEEE, 2014.

[13] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[14] Spiros Papadimitriou and Jimeng Sun. Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 512–521. IEEE, 2008.

[15] Efstratios Rappos and Stephan Robert. Predictive caching in computer grids. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 188–189. IEEE, 2013.

[16] Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[17] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[18] Mingming Sun, Hang Zhuang, Xuehai Zhou, Kun Lu, and Changlong Li. Hpso: Prefetching based scheduling to improve data locality for mapreduce clusters. In *Algorithms and Architectures for Parallel Processing*, pages 82–95. Springer, 2014.

[19] Bo Wang, Jinlei Jiang, and Guangwen Yang. mpcache: Accelerating mapreduce with hybrid storage system on many-core clusters. In *Network and Parallel Computing*, pages 220–233. Springer, 2014.

[20] Tom White. *Hadoop: the definitive guide: the definitive guide.* " O'Reilly Media, Inc.", 2009.

[21] Jiong Xie, FanJun Meng, HaiLong Wang, HongFang Pan, JinHong Cheng, and Xiao Qin. Research on scheduling scheme for hadoop clusters. *Procedia Computer Science*, 18:2468–2471, 2013.

[22] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 198–207. IEEE, 2009.

[23] Matei Zaharia, Dhruba Borthakur, J Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55*, 2009.

[24] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[26] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[27] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Accelerating mapreduce with distributed memory cache. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 472–478. IEEE, 2009.

[28] Xiaohong Zhang, Zhiyong Zhong, Shengzhong Feng, Bibo Tu, and Jianping Fan. Improving data locality of mapreduce by scheduling in homogeneous computing environments. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 120–126. IEEE, 2011.