Predoop: Preempting Reduce Task for Job Execution Accelerations

Yi Liang[1(✉)], Yufeng Wang[1], Minglu Fan[1], Chen Zhang[1], and Yuqing Zhu[2]

[1] Department of Computer Science, Beijing University of Technology, Beijing, China
{yliang, yufengwang, fanminglu}@bjut.edu.cn

[2] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy Sciences, Beijing, China
Zhuyuqing@ict.ac.cn

Abstract. Map/Reduce is a popular parallel processing framework for data intensive computing. For overlapping the Map task's execution phase and the Reduce task's intermediate data fetching and merging phase, existing Map/Reduce schedulers always pre-launch the Reduce task at the specific threshold where its map tasks have been launched, and this pattern incurs the occupation of the consuming resources of the reduce task during its idle time on waiting for fetching the intermediate data from map tasks. To address this issue, we propose an extension version of Hadoop map/reduce framework, called Predoop, in this paper. The basic idea of Predoop is to preempt the reduce task during its idle time and allocate the released resource to the map tasks on schedule. To achieve this goal, first, we introduce the preemptive mechanism for reduce tasks and map tasks respectively to enable Map/Reduce tasks to be preempted or resumed with correct status; second, we adopt the preempting-resuming model for the reduce task with the consideration of the progress of Reduce task data fetching & merging and the Map task execution so as to determine the timing of Reduce task preemption and resuming; third, we introduce the preemption-aware task scheduling strategy to allocate the released resources to the on-schedule Map tasks with the consideration of data locality. Experimental result demonstrates that Predoop outperforms Hadoop on various workload and the average job turnaround time can be reduced by maximum of 66.57 %.

Keywords: Map/Reduce · Intermediate data dependency · Preemption · Task scheduling

1 Introduction

Map/Reduce is a new parallel processing framework for programming the commodity computer clusters to perform the large-scale data processing [1]. The scheduling granularity for each scheduled job is on the task level in the Map/Reduce framework [1]. Once a map or a reduce task is launched, it will stay in active and occupy its allocated computing resource, such as the memory space or the CPU slots. In general, each map task always processes one data block of total job input data sets and each

Reduce task would process the output data from all of map tasks corresponding to its processing partition. We call this data dependency between the Map task and the Reduce task as intermediate data dependency.

Due to the intermediate data dependency between map tasks and reduce tasks, existing map/reduce schedulers always pre-launch the Reduce task at the specific threshold (often 10 %) where its map tasks have been launched. Under the ideal situation, this setting can overlap the map task's execution phase and the reduce task's intermediate data fetching and merging phase [2]. However, due to the asymmetry in the completion time of map tasks of a job (because of the difference in the launching time or the uneven data processing progress), the Reduce task often stays in idle and consumes little of its occupied resources at the stage where part of its dependent map tasks have been data-fetched while others still on execution. The reduce task's idle time contributes much to the inefficient resource utilization, and hence, pulls down the efficiency of map/reduce job execution. Our examination shows that, running 20 WordCount map/reduce jobs on a 12-node cluster, the idle time of a reduce task can be, by average, 44.5 % to its total execution time and 23.3 % to its job's total execution time. The situation goes worse when the resource competition becomes more intensive (that is, more jobs in the workload) [3].

To address this performance issue, we present an extension version of Hadoop map/reduce framework, called Predoop [4]. The motivation of Predoop is to preempt the idle reduce tasks to mitigate the idle time, and allocate the resources to map tasks on schedule to accelerate the job execution. To achieve this goal, Predoop introduces a preemption model for reduce tasks to determine the time point of suspending or resuming the reduce task. Based on the preemption model, Predoop adopts a preemption-aware task scheduling strategy to guarantee that the on-schedule map tasks are allocated with those released resources. Further, Predoop integrates the enabling preemptive mechanisms for reduce tasks and map tasks to make sure that the preemption model and task scheduling are practical. The main contributions of Predoop are as follows:

(1) The preempting-resuming model for the reduce task. The definition of the time point to preempt/resume reduce task is the most fundamental factor in the reduce task preemption solution. We introduce two quantitatively estimation models—preempting model and resuming model, to determine the reduce task preempting and resuming occasion. To improve the preemption efficiency, the preempting model is designed based on the ratio of the progress of the reduce task's data fetching & merging to the map task's execution; the resuming model is design based on the ratio of the number of completing map tasks after preemption to total number of map tasks.

(2) Preemption-aware task scheduling for the reduce task preemption. Based on the preempting-resuming model, we adopt the preemption-aware task scheduling to schedule the preempted resources in high priority. We design a new scheduling strategy for preempted resources, which allocates these resources to map tasks and avoids the fragmentized execution of the map task due to its consuming resource reclaimed by the resuming reduce tasks frequently.

(3) Preemptive mechanisms for map tasks and reduce tasks. By recording the boundary of <key, value> pair processed by a map task, and the boundary of map task that has been intermediate data-fetched, the preemptive task mechanisms can assure the map and reduce task be resumed with the correct status and not losing previous works.
(4) We have conducted the performance evaluation for Predoop with two famous benchmark suites: SWIM and BigDataBench [5, 6]. The experimental results demonstrate that Predoop outperforms the native Hadoop on both the synthetic workloads (SWIM) and real world workloads (BigDataBench). It can reduce the average turnaround time of map/reduce jobs by up to 66.57 %.

The following sections are organized as follows: Sect. 2 describes the preempting-resuming model of reduce tasks; Sect. 3 present the preemption-aware task scheduling in Predoop; Sect. 4 introduces the preemptive mechanisms for the reduce task and map task respectively; Sect. 5 analyzes the experimental results; Sects. 6 and 7 present the related work, the conclusion and the future work of this paper respectively.

2 Preempting-Resuming Model of Reduce Tasks in Predoop

As described before, the main idea of Predoop is to preempt a reduce task during its idle time in the fetching and merging phase and allocate its released computing resources to some map tasks to be scheduled. Features of the preempting-resuming model of reduce tasks are to decide the time point to perform the preempting operation on a reduce task, and the time point to resume it and reallocate the computing resource it occupied before.

2.1 Preempting Model of Reduce Task

On designing the preempting model, we take two factors into consideration. One is the start point of a reduce task's idle time. It is obviously the candidate time point to preempt a reduce task. The second is the length of a reduce task's idle time. This is for that idle time of the reduce task could be too short to cover the time cost of the backfilling map task's deployment so that the benefit of utilizing the preempted resources will be overthrown. To make the most use of the limited idle time, Predoop determines the candidate time point of the reduce task preempting in an advance way by estimating the start point and the length of reduce task's idle time periodically. Once the time length is long enough, the corresponding start time will be chosen as the candidate preempting time.

For each preempting decision making, the starting point of a reduce task‘s idle time can be calculated out with the factor of ***Remaining Fetch & Merge time*** from the deciding time. The ***Remaining Fetch & Merge time*** can be defined as the remaining time that a reduce task needs to complete fetching and merging the intermediate data generated by the map tasks that has been completed. Due to that, in Predoop, a reduce task performs the data fetching & merging with a thread pool and fetches the intermediate data one group after another. The Remaining Fetching & Merging time is estimated as T_{rfm} through the following function.

$$T_{rfm} = T_{fm} + \left\lceil \frac{N_{wait}}{k} \right\rceil \times T_{iter} \tag{1}$$

Where, T_{fm} stands for the average time that a reduce task spends to fetch and merge the intermediate data from a map task; N_{wait} is the number of map tasks that have been completed with their intermediate data not fetched; k is the number of map tasks that a reduce task can start to fetch their intermediate data roughly at the same time and run in a single wave; T_{iter} is the average time interval between two successive data fetching waves for a reduce task. In predoop, the k is initially set as 1 and the T_{iter} is set as the minimal remaining execution time among all active data fetching threads. As a reduce task makes progress in its execution, we dynamically update the T_{fm}, k and T_{iter} accordingly.

The figure out the length of a reduce task's idle time, we need to estimate the end time point of this time period. In predoop, the end time of a reduce task' idle time can be decided with the factor of ***Remaining execution time of map task***, which the reduce task depends on, from the decision making time point. In other word, once these map tasks finish execution, the reduce task may be reactive to fetch and merge the new-generated intermediate data. The ***Remaining execution time of map task*** is estimated as T_{rm} through the following function.

$$T_{rm} = \frac{1 - p_{mt}}{p_{mt}} \times T_{mex} \tag{2}$$

Where, p_{mt} stands for the execution progress of a map task. P_{mt} can be calculated out during a map task's execution according to the proportion of data that have been processed. T_{mex} is the map task's total execution time since its beginning.

The preempting model of reduce task finally decides the candidate preempting time point of a reduce task according to the following condition.

$$\frac{Min\left\{T_{rm1}, T_{rm2}, \cdots, T_{rmn}\right\} - T_{rfm}}{T_{mte}} \geq D_p \tag{3}$$

Where, the set of T_{rmi} $(1 \leq \mathrm{i} \leq \mathrm{n})$ stands for the remaining execution time of all map tasks that the reduce task depends on; T_{mte} is the average execution time of the completed map tasks that the reduce task depends on; D_p is a threshold which indicates to what extend is the reduce task's idle time long enough to perform the preempting operation.

In predoop, the periodical prediction of a reduce task's preempting time will stop when a candidate time point is generated, and restart when the reduce task resumes from a preemption. To compensate the inaccuracy in the estimation, the preempting operation on a reduce task will be carried out immediately once the reduce task shifts to the idle state ahead of the candidate preempting time point. On the other hand, if the reduce task's data fetching & merging operation on its depending map tasks, which has completed on the preempting decision making time point, does not finish on the candidate preempting time point, the preempting operation of the reduce task will be postponed until all fetching & merging operations complete.

2.2 Resuming Model of Reduce Task

In predoop, a preempted reduce task can only be resumed when there are some map tasks that it depends on have been completed during its preemption. The resuming model determines the resuming of a reduce task only if the following two conditions are satisfied.

Condition 1:

$$\frac{N_{map\_c} - N_{map\_f}}{N_{map}} \geq D_r \tag{4}$$

Where, N_{map_c} stands for the number of completed map tasks that a reduce task depends on; N_{map_f} is the number of map tasks that a reduce task depends on and have completed with the generated intermediate data not fetched; N_{map} is the total number of map tasks that the reduce task depends on; D_r is a threshold.

Condition 2: All map tasks allocated with the preempted computing resource of the reduce task are not in the intermediate data partition phase.

In a word, Condition 1 guarantees that only when the number of its depending map tasks has accumulated to be large enough, the preempted reduce task can be resumed. Condition 1 is established to prevent the frequent preempting/resuming of reduce tasks and make the reduce task fetch the intermediate data in a bundle way. Because the partition phase is the last phase of map task execution and leads to heavy disk I/O cost, Condition 2 makes the restriction that the intermediate data partitioning operation can be performed in all-or-nothing way, so as to simplify the preemption operation of map task, and make sure that the disk I/O cost caused by the data partitioning can be returned with some progress in the map task execution.

3 Preemption-Aware Task Scheduling in Predoop

The most distinguished feature of task scheduling in Predoop is to allocate the resources released from the preempted reduce tasks (we call them as ***preempted resource***) to the on-schedule map tasks. Similar to Hadoop, task scheduling in Predoop is triggered with the 'asking for the new task' heartbeat message sent from a computing node with the available resources information enclosed. The scheduler performs the task scheduling on the preempted resources with the following three rules:

Rule 1: The allocation of preempted resource is prior to the regular resource.

Where, the regular resource refers to the available resource released by the map/reduce task that completed or failed normally.

Rule 2:

$$\neg\exists t(t \in PR \wedge (Aloc(t) \cap GR \neq \varphi \vee Aloc(t) \cap PR \neq \varphi)) \tag{5}$$

Where, PR stands for the preempted reduce task set in Predoop; GR stands for the non-preempted reduce task set; the function *Aloc()* can be expressed as Aloc:T –> T, where T is the total task set in Predoop. Function *Aloc(t)* defines the task set that allocated with the computing resource that task t has released when completed or suspended.

Rule 3:

$$\neg\exists t\left(t \in MP \wedge t \in Aloc(pr_i) \wedge t \in Aloc(pr_j),\ \ pr_i \in PR, pr_j \in PR, pr_i \neq pr_j\right) \tag{6}$$

Where, MP stands for the map task set in Predoop.

Rule 1 is established for that the use of preempted computing resource is highly time sensitive (only available during the idle time of a reduce task). Rule 2 guarantees that the preempted resource can only be allocated to the map task. This is for that idea of Predoop is to preempt the reduce task only when it is idle. However, once a reduce task is allocated with the preempted resource, it may be interrupted during its data fetching when the corresponding preempted reduce task needs to be resumed and reclaim the resource. Rule 3 prevent that the resources allocated to a map task is released from multiple reduce tasks. This is to avoid the scenario where a map task 'gathers' the resource fragment from multiple suspended reduce tasks and leads to its frequent interruption because any of those reduce tasks needs to be resumed.

Based on the three rules, Predoop queues the map/reduce job in FIFO (First In First Out) way and assigns the preempted resource to map tasks with the consideration of node-level, rack-level and offSwitch-level data locality in sequence. On the other hand, among the map tasks with node-level data locality, Predoop chooses the task, that has been preempted because their consuming resources are reclaimed by the reduce tasks, in prior. This is because that the more preempted map tasks accumulated during a job's execution, the larger amount of intermediate data its reduce tasks need to fetch later, and hence increases the risk of network burst. For the regular available resources, Predoop inherits the task scheduling strategy from Hadoop.

4 Preemptive Task Mechanism in Predoop

In predoop, map tasks and reduce tasks are applied with different preemptive task mechanisms on their consuming resources preempted or reallocated.

As described above, the preemption of reduce task can only occur when it finishes the intermediate data fetching and merging from part of its dependent map tasks during its shuffle time. During its data fetching & merging phase, the reduce task 'pull' the intermediate data from multiple dependent map tasks in a parallel way and store them into data segments in memory or on disk according to the data size. The data segments are then merged into the larger segment and stored into the disk. Figure 1(a) shows the

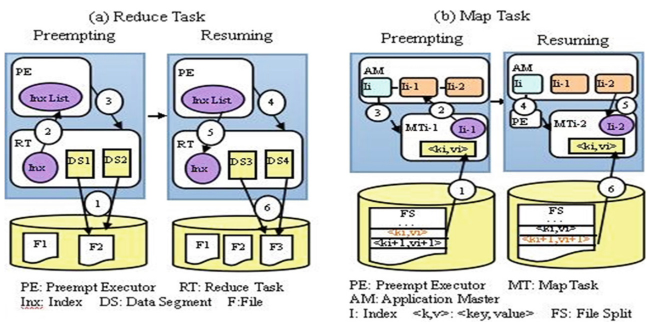
(a) Reduce Task
Preempting
Resuming
PE
Inx List
RT
Inx
DS1
DS2
DS3
DS4
F1
F2
F3
(b) Map Task
AM
Ii
Ii-1
Ii-2
MTi-1
MTi-2
<ki,vi>
FS
<ki+1,vi+1>
PE: Preempt Executor RT: Reduce Task
Inx: Index DS: Data Segment F:File
PE: Preempt Executor MT: Map Task
AM: Application Master
I: Index <k,v>: <key, value> FS: File Split

Fig. 1. Preemptive mechanism for reduce and map tasks

preemptive mechanism of reduce task in Predoop. In Predoop, each reducae task holds a index during it execution. The index records the dependent map tasks that have or haven't been completed yet, and the location of data segment files belonging to the reduce task. The index is updated dynamically when the reduce task is in progress. On the preemption, the reduce task first stops the updating of the index, completes the fetching and merging of data output from all completed map tasks, and flushes all data segments in memory into the disk (step 1). The reduce task then backups its index information to the Preempt Executor (step 2). Finally the Preempt Executor preempts the reduce task (actually kill the process of the reduce task) (step 3). When resuming the reduce task, the Preempt Executor first restarts the reduce task (step 4). Once restarted, the reduce task gets the index from Preempt Executor and makes clear of the map tasks to fetch the intermediate data (step 5). The reduce task then pulls the map output and merges into new data segment files (step 6).

In Predoop, the map task can only be preempted during its map phase. During the map phase, the map tasks read and process the <key, value> pair from the distributed file system HDFS in sequence. Figure 1(b) shows the preemptive mechanism of map task in Predoop. In Predoop, each map task can only be preempted at the end of each <key, value> pair and the index of the last processed <key, value> pair needs to be recorded. On preemption, the map task first finishes processing the <key, value> pair on hand. The map task then records the index of the last processed pair to its Application Master. Application Master resets the status of the map task as 'on schedule' (step 2). Finally, Application Master preempts the map task (actually kill it) via Preempt Executor resided on the same node as the map task (step 3). On resuming, the Application Master restarts the map task (step 4). The map task gets the index of last processed <key, value> pair information from Application Master (step 5). The map task then restarts the data processing from the <key, value> pair next to the last processed one.

5 Performance Evaluation

In this section, we present a systematic performance evaluation of Predoop. We compare the performance of Predoop to YARN (a new version of Hadoop) with the FIFO scheduler. This is for that FIFO is the fundamental of others schedulers. When porting to other scheduler, the advantage of preemptive scheduling may be amplified due to the fact that there are multiple job queues in other schedulers and concurrent preemptions can be conducted.

5.1 Experimental Methodology

We first conduct a systematic performance evaluation of Predoop with a diverse sets of workloads, including load-shrinking and load-amplifying workload from BigData-Bench benchmark, and the mix workload from Swim [5, 6]. Further, we study the sensitivity of Predoop performance to the configuration of two thresholds in the preemption model (Dp and Dr), due to the fact that various configurations result in the different occasion and frequency of reduce task preemption. Finally, we evaluate the scalability of Predoop.

Experiments are conducted in a cluster of 13 nodes. One node is dedicated as both the ResourceManager and NameNode. Each node is equipped with two Intel(R) Pentium(R) 4 cpus, 3 GB memory and one 160 GB SATA hard driver. On the YARN configuration, we configure totally 2 GB memory per node and assign 1024 MB memory for each Application Master. The HDFS block size is set as 64 MB as default.

Two benchmarks are employed. One is BigDataBench, which provides the real world ap-plication workloads with real world data sets, and we choose two single-job workloads from it: WordCount and Sort. WordCount represents the workload category that includes map/reduce job which generates small amount of intermediate data so that the reduce tasks have lighter load than map tasks (We called them load-shrinking workload). Sort represents the workload category which generates large amount of intermediate data so that reduce tasks have much heavier load than map tasks (We call them load-amplifying application). The other benchmark is SWIM. SWIM can generate the synthetic workload for diverse size of Hadoop cluster according to the trace of Facebook product map/reduce platform. We add the sleep() to the map and reduce task function body so as to guarantee the execution time of map/reduce task in accordance with the heavy-tail distribution [7].

We choose Average Turnaround Time as the main evaluation metric. Average Turnaround Time is the most typical metric to reflect the efficiency of a scheduler of the system.

5.2 Result for the Single-Application Workloads

We first conduct the experiment on single-application workload with the input data size of 8 GB, 10 GB, 12 GB, 14 GB, 16 GB. For each job, we set the reduce task number as 8. The required memory amount of each task is set as 1024 MB as default. The thresholds D_p and D_r in the preemption model of reduce task are set as 20 % and 40 % respectively.

Figure 2 shows that Predoop outperform YARN on the execution of single-application workload with various data sizes. For the Sort workload, the average job turnaround time is reduced by 29.5 % on average and 49.07 % by maximum. Predoop

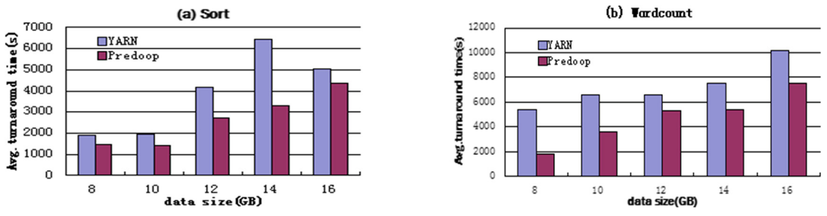

Fig. 2. Average turnaround time of single-application workloads

also achieves performance improvement on the load-shrinking workload, like Wordcount. The average job turnaround time is reduced by 37.24 % on average and 66.57 % by maximum.

According to the statistics, due to the preemptive mechanism, Predoop minifies the reduce task's idle time by 95.86 %–99.93 % compared to YARN, and allocates the preempted resources to map tasks on schedule. This improvement contributes much to the promotion of the job turnaround time. On the other hand, the performance result shows that Predoop achieves better performance promotion on the load-shrinking workload (like Wordcount) than the load-amplifying workload. This may be for that when map tasks have heavier load and output smaller intermediate data, there will be more chances for the reduce tasks to complete one round of data fetching & merging quickly and leave more idle time to be preempted during its waiting for the next round of map task completion.

5.3 Results for the Mix Workloads

To evaluate the performance of Predoop in the shared map/reduce cluster, we use four mix workloads generated by SWIM. Among these mix workload, the proportion of load-amplifying job varies from 6 % to 8.7 %, which represents the typical mixture ratio in the product map/reduce clusters (like Facebook).The thresholds D_p and D_r in the preemption model of reduce task are also set as 20 % and 40 % respectively. To simulate the memory resource contention in the shared map/reduce cluster, we vary the memory requirement of each map and reduce task as 512 MB, 1 GB (default set in YARN), and 1.5 GB (Table 1).

Figure 3 shows that Predoop outperforms YARN on the mix workload experiment. The average job turnaround time is reduced by up to 49.85 %. According to the statistics, the drop rate of the average reduces task's idle time keeps relatively stable

Table 1. Characteristics of mix workloads

	Bin1	Bin2	Bin3	Bin4
Job number	120	150	180	200
Total size of map input data (GB)	46.66	64.19	72.32	82.94
Total size of intermediate data (GB)	6	6.25	6.47	6.58
Total size of reduce output data (GB)	1.36	1.44	2.26	7.19

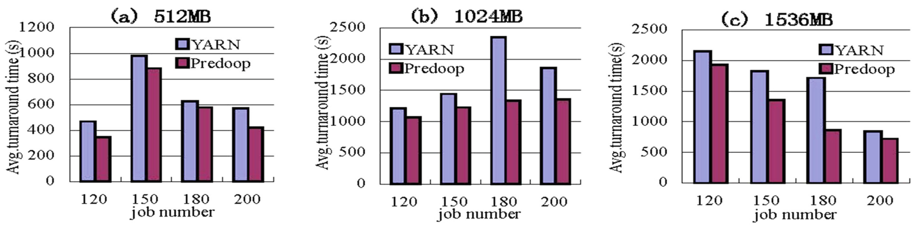

Fig. 3. Average turnaround time of mix workloads

under the different memory requirements (varying from 89 % to 90.7 %). However, the drop rate of the average job turnaround time goes up from 18.7 % to 25 %, with the memory requirement per task increasing (In another word, the resource contention more intensive). This is due to the fact that, with the resource contention, jobs with larger map task size may have higher risk to launch map tasks in batch. The preemptive scheduling can preempt the idle reduce tasks, contribute their occupied resource to help the on-schedule map tasks hold their required memory resource more quickly, and hence, accelerate the job completion. The statistic result shows that the performance improvement of the jobs with larger map task size contributes much to the increasing drop rate.

5.4 Performance Sensitivity to the Threshold Configurations

As described in Sect. 3, the preemption model of reduce tasks is designed with two threshold parameters: Dp and Dr. Performance of Predoop may be sensitive to the threshold configuration due to that these two thresholds control the occasion and frequency of reduce task preempting and resuming and may incur extra cost on the task status switch. To evaluate the performance sensitivity, we choose the mix workload

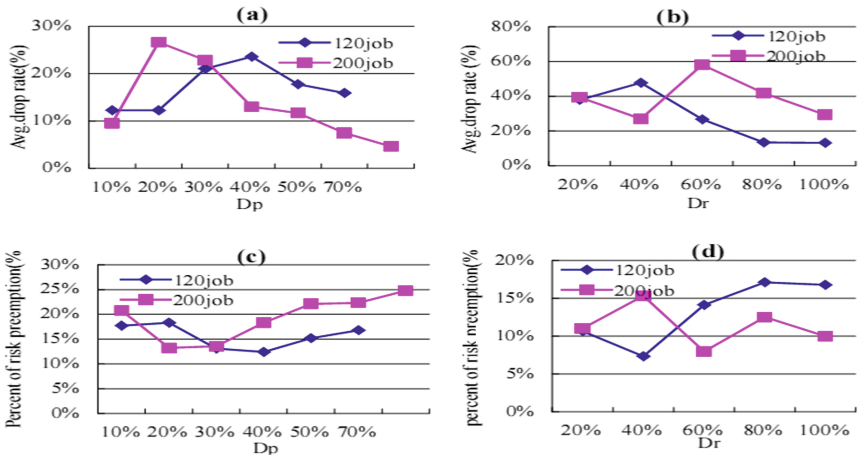

Fig. 4. Performance sensitivity to threshold configuration

with the job number of 120 and 200 and conduct the experiments by varying the threshold configuration. We vary Dp in preempting model as 10 %, 20 %, 30 %, 40 %, 50 %, 60 %, 70 %, and vary Dr in resuming model as 20 %, 40 %, 60 %, 80 %, 100 %.

Figure 4(a) and (b) shows the variation of average drop rate on turnaround time by Predoop. We find that for D_p, the best performance can be achieved when setting the threshold as 30 %–40 % for the 120-job mix workload and 20 %–30 % for the 200-job mix workload. To make it clear, we count the percent of risk preemption among all the reduce task preemption. The ***risk preemption*** is defined as the reduce preemption that leave the reduce task in preempting for less than 10 % of its dependent map tasks' average execution time. Figure 4(c) demonstrates that the variation of percent of risk preemption is quite in accordance with that of drop rate on trunaround time. This is due to the fact that setting the threshold too small will lead the reduce task to be preempted frequently and provide the preemption time not long enough to accommodate the efficient execution of map tasks, but only induce the extra map&reduce task start/stop cost. When setting the threshold too large, the reduce task will delay its preemption and keep it in the idle state so as to shorten the time period that map tasks consume the preempted resource.

When varying the configuration of D_r, we find the similar performance variation pattern as that of D_p in Fig. 4(b) and (d). The cause is also similar. When the threshold is set too small, the reduce task needs to be resumed quite frequently and leaves too short preemption time. When the threshold is set too large, the reduce task will stay in the preemption even when there is enough fetching data generated, so that delay the completion of reduce tasks.

5.5 Scalability

We evaluate Predoop's performance scalability according to the cluster size. We generate five groups of workloads for the cluster size of 4, 6, 8, 10, 12. For each group, we generate three workloads with 120 jobs each. For each group, we calculate the average job turnaround time of the corresponding three workloads.

Figure 5 shows that with the typical synthetic workloads, Predoop outperforms YARN on diverse cluster sizes. The average turnaround time is reduced by the maximum of 46.29 %, and by the minimum of 20.98 %. According to the statistics, the average reduce task idle time is cut down by maximum of 98.37 % and by minimum of 91.28 %.

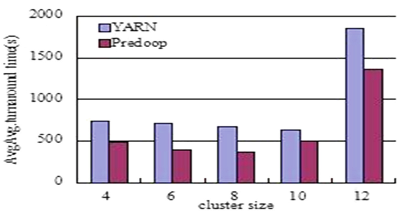

Fig. 5. Average turnaround time with diverse cluster size

6 Related Works

Many task schedulers have been proposed for map/reduce cluster over the past several years to pursue either the fairness among jobs or maximization of the job execution performance. To address the fairness issue, Hadoop, the most popular open-source implementation of Map/Reduce framework, provides three task scheduler: FIFO Scheduler, Fair Scheduler, Capacity Scheduler [3]. In [8], based on these fundamental task scheduling, the fair scheduler is improved in Hadoop by postponing the execution of head-of-queue tasks when the computing node to be allocated doesn't hold its processing data locally. Quincy introduced a min-cost flow algorithm to achieve the tradeoff between the fairness and data locality [9]. However, most fairness-centra schedulers don't adopt the preemptive mechanism and cannot prevent the long job monopolizing the system capacity or the significant resource waste.

There are several ways to maximize the job performance: (1) overlapping or subdividing some phases in a map/reduce job so as to overlap the execution of phases that utilize different resources of CPU and disk i/o. Works in [10] split the reduce task of a map/reduce job into the data copy task and the data computing task. However, it can not resolve the resource waste issue during the data copy task's idle time; (2) reducing the i/o cost by the aware of data locality or the network status. References [11, 13] present a data locality-aware and skew-aware reduce task scheduler to shorten the reduce task execution time; Maestro improves the locality of map task execution by keeping track of the data chunk and its replication location [12]. Reference [14] proposed the communication-aware placement and scheduling of map tasks and predictive load-balancing of reduce tasks so as to reduce the data i/o cost during the job execution. All the four works focus on the optimization of data i/o cost, but ignore the data dependence between map and reduce tasks; (3) remedying the outlier of map/reduce task execution particularly in the heterogonous environment. Mantri can identify the outlier in the map/reduce clusters by real-time monitoring task execution and restart the outliers on the node chosen with network awareness [15]. Though Mantri conserves some valuable work for the outlier task, the preemptive mechanism is not introduced in it. What's more, the performance optimization of the regular tasks is not Mantri's focus; (4) predicting the execution of map/reduce task and adjust the resource allocation dynamically so as to meet the SLA. ARIA conducts the job profiling and designs the map/reduce performance model to estimate the amount of resource a routinely executed job required to complete within the deadline [16]. Reference [17] adopted the preemptive mechanism for reduce task and designs the task scheduling for the fairness issue. Hence, the scheduling algorithm is totally different from that in Predoop.

7 Conclusion and Future Work

In this paper, we propose an extended map/reduce framework called Predoop. Predoop aims at solving the issue that the reduce task occupies the allocated resource during its idle time when waiting for the intermediate data fetching from its dependent map tasks, which lowers the job performance. Idea of Predoop is to preempt the reduce task during its idle time and allocate the released resource to the map tasks on schedule. To achieve this goal, Predoop adopts the effective preemptive mechanism for both reduce and map task, and defines the preempting-resuming model of reduce tasks with the consideration of the progress of reduce task data fetching & merging and the map task execution. Based on the preempting-resuming model, a preemptive task scheduling strategy is present to allocate the preempted resources to map tasks concerning the data locality. Experimental results demonstrate that Predoop outperforms Hadoop for the load-amplified workload, the load-shrinked workload and the mix workload. The average job turnaround time is promoted by the maximum of 66.57 %. The ongoing work includes: (1) improving the preempting-resuming model for the map/reduce job that has asymmetric processing time on multiple data elements; (2) the online adjustment of the threshold in the preemption model.

Acknowledgements. This work is supported by NSFC projects (Grants No. 60933003 and 61202075) and BNSF project (Grant No. 4133081).

References

1. Chen, S., Schlosser, S.: Map-reduce meets wider varieties of applications. Technical report, IRP-TR-08-05 (2008)
2. Dean, J., Ghemawat, A.: MapReduce: simplified data processing on large clusters. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI04), May 2004, pp. 137–150. ACM Press (2004)
3. Wang, Y.: Data dependency in map/reduce cluster. Technical report, BJUT-TR-14-01 (2014)
4. Apache Hadoop. http://hadoop.apache.org/
5. https://github.com/SWIMProjectUCB/SWIM/wiki
6. Wang, L., Zhan, J., Luo, C., Zhu, Y.: Bigdatabench: a big data benchmark suite from internet services. In: Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA-14), pp. 21–32. ACM (2014)
7. Chen, Y., Alspaugh, S., Katz, R.: Interactive query processing in big data systems: a cross-industry study of MapReduce workloads. In: Proceedings of the 38th International Conference on Very Large Data Bases (VLDB 2012), pp. 12–23. ACM (2012)
8. Zaharia, M., Borthankur, D., Sarma, J.S.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the European Conference on Computer Systems (EuroSys'10), pp. 265–278. ACM (2010)
9. Isard, M., Prabhakaran, V., Currey, J.: Quincy: fair scheduling for distributed computing clusters. In: Proceedings of the ACM Symposium on Operating Systems Principles (SIGOPS'09), pp. 261–276. ACM Press (2009)
10. Zaharia, M., Borthakur, D., Sarma, J.S., et al.: Job scheduling for multi-user map/reduce clusters. Technical report, UCB-EECS-2009-55 (2009)
11. Hammoud, M., Rehman, M. S., Sakr, M.F.: Center-of-gravity reduce task scheduling to lower MapReduce network traffic. In: International Conference on Cloud Computing (CLOUD), pp. 49–58. IEEE (2012)
12. Ibrahim, S., Jin, H., Lu, L., et al.: Maestro: replica-aware map scheduling for MapReduce. In: International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 435–442. ACM/IEEE (2012)

13. Tan, J., Meng, S., Meng, X., et al.: Improving ReduceTask data locality for sequential MapReduce jobs. In: International Conference on Computer Communications (INFOCOM), pp. 1627–1635. IEEE (2013)
14. Ahmad, F., Chakradhar, S.T., Raghunathan, A., Vijaykumar, T.N.: Tarazu: optimizing MapReduce on heterogeneous clusters. In: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12), pp. 61–74. ACM (2012)
15. Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A.G., Stoica, I., Lu, Y.: Reining in the outliers in map-reduce clusters using mantri. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10), pp. 18–28. ACM (2010)
16. Verma, A., Cherkasova, L., Campbell, R.H.: ARIA: automatic resource inference and allocation for MapReduce environments. In: International Conference on Autonomic Computing (ICAC), pp. 235–244. ACM (2011)
17. Wang, Y., Tan, J., Yu, W.: Preemptive ReduceTask scheduling for fair and fast job completion. In: Proceedings of the 10th International Conference on Automatic Computing (ICAC-13), pp. 45–56. ACM (2013)