Dong JQ, He ZH, Gong YY *et al.* SMART: Speedup job completion time by scheduling reduce tasks. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(4): 763–778 July 2022. DOI 10.1007/s11390-022-2118-5

SMART: Speedup Job Completion Time by Scheduling Reduce Tasks

Jia-Qing Dong[1] (董加卿), Ze-Hao He[2] (何泽昊), Yuan-Yuan Gong[2] (龚媛媛), Pei-Wen Yu[2] (于沛文)
Chen Tian[2] (田　臣), *Senior Member*, *IEEE*, *Member*, *CCF*, *ACM*, Wan-Chun Dou[2,*] (窦万春)
Gui-Hai Chen[2] (陈贵海), Nai Xia[2] (夏　耐), and Hao-Ran Guan[3] (管浩然)

[1] *State Key Laboratory of Media Convergence and Communication, Communication University of China Beijing 100024, China*

[2] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*

[3] *School of Computer Science, The University of Sydney, Sydney NSW 2006, Australia*

E-mail: jiaqing@cuc.edu.cn; {hezehao, mf1933022, pavin}@smail.nju.edu.cn
{tianchen, douwc, gchen, xianai}@nju.edu.cn; hgua5212@uni.sydney.edu.au

Received December 27, 2021; accepted June 29, 2022.

Abstract Distributed computing systems have been widely used as the amount of data grows exponentially in the era of information explosion. Job completion time (JCT) is a major metric for assessing their effectiveness. How to reduce the JCT for these systems through reasonable scheduling has become a hot issue in both industry and academia. Data skew is a common phenomenon that can compromise the performance of such distributed computing systems. This paper proposes SMART, which can effectively reduce the JCT through handling the data skew during the reducing phase. SMART predicts the size of reduce tasks based on part of the completed map tasks and then enforces largest-first scheduling in the reducing phase according to the predicted reduce task size. SMART makes minimal modifications to the original Hadoop with only 20 additional lines of code and is readily deployable. The robustness and the effectiveness of SMART have been evaluated with a real-world cluster against a large number of datasets. Experiments show that SMART reduces JCT by up to 6.47%, 9.26%, and 13.66% for `Terasort`, `WordCount` and `InvertedIndex` respectively with the Purdue MapReduce benchmarks suite (PUMA) dataset.

Keywords job scheduling, job completion time, MapReduce, Hadoop

1 Introduction

Distributed computing systems have been widely used as the amount of data grows exponentially with the rapid development of various network-based applications. In order to solve the problem of storage and processing of petabyte-level data, distributed computing systems such as Hadoop, Spark and Hive emerge as the times require. They have high reliability, high scalability, high fault tolerance and efficiency, and can support processing very large datasets across server clusters. These systems meet the requirements of large-scale data processing, such as advertising recommendation, log processing and user behavior analysis. How to improve the performance of these systems has become a hot issue in both industry and academia. It is very important to find a good scheduling mechanism for these systems to speed up these jobs and carry out more efficient processing.

Job completion time (JCT) is a major metric to evaluate such distributed computing systems [1–4]. In this paper, we focus on minimizing JCT of distributed computing systems.

Data skew, which can compromise the performance of such distributed computing systems, is not uncommon [5, 6]. As an example, we observe that the

Regular Paper
Special Section of Xia Peisu Young Scholars Forum 2021
This work was supported by the National Key Research and Development Project of China under Grant No. 2020YFB1707600, the National Natural Science Foundation of China under Grant Nos. 62072228, 61972222 and 92067206, the Fundamental Research Funds for the Central Universities of China, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

*Corresponding Author

input data of reduce tasks (i.e., the intermediate output data generated by the end of map tasks) is skewed during the daily usage of Hadoop. However, we find that the default scheduling mechanism for reduce tasks in Hadoop does not take the data skew into consideration, and simply schedules all reduce tasks in a first-in-first-out (FIFO) manner. Here comes the question.

Can we leverage this feature to design a good scheduling mechanism to speed up the overall job completion time?

The module we propose in this paper, named as SMART, answers this question affirmatively.

We envision that scheduling the reduce tasks with the largest-first mechanism will be helpful to improve the performance. Intuitively, a good scheduler should overlap large reduce tasks with smaller tasks as many as possible. It can effectively reduce JCT by scheduling reduce tasks with longer execution time first.

We use the example shown in Fig. 1 to illustrate the issues in scheduling reduce tasks with skewed input data. In Fig.1, each slice stands for a reduce task, with the slice label representing the task duration. All the reduce tasks belonging to the same job execute the same part of the code. Therefore, if the hardware devices are the same, it can be regarded that the total processing time will be proportional to the data size. Suppose that there are three reduce tasks, the time required for each task is 3 T, 2 T, and 1 T respectively, where T stands for the basic time unit. We mark the three reduce tasks as T_3, T_2 and T_1 respectively. In Hadoop, all scheduled reduce tasks will be queued waiting for slots to be released. In the example, there are three slots released at 1 T, 2 T, 3 T respectively for reduce tasks to run.

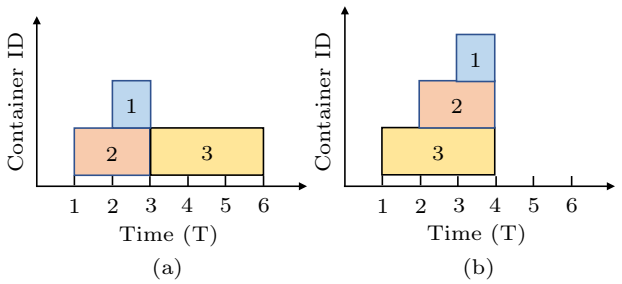


Fig. 1. Example of reduce task scheduling, where largest-first takes 40% less time than FIFO. (a) FIFO. (b) Largest-first scheduling.

Fig.1 shows the task execution details of both FIFO scheduling and largest-first scheduling. The scheduling order of FIFO is $\{T_2, T_1, T_3\}$, and the scheduling order of largest-first is $\{T_3, T_2, T_1\}$. The result shows that the execution time of largest-first is 3 T, while the execution time of FIFO is 5 T. In this example, largest-first scheduling takes 40% less execution time than FIFO.

In this paper, we propose SMART which can effectively improve the overall performance of distributed computing systems by enforcing largest-first scheduling in the reducing phase. We implement SMART in Hadoop, which is one of the most widely used distributed frameworks for massive data processing, with just minimal modification to the original Hadoop source code. It is worth noting that SMART is not limited to Hadoop. SMART can bring benefits to all distributed computing systems with the MapReduce paradigm where data skew exists.

The contributions of this paper can be summarized as follows.

• We propose a reduce task size prediction method based on part of the completed map tasks. Experimental results demonstrate that the proposed method is robust and effective.

• We formally define and analyze the reduce task scheduling problem and propose to apply the largest-first scheduling mechanism for the reducing phase in MapReduce jobs.

• We design and implement SMART on Hadoop, with minimal modification to native Hadoop source code. The implementation shows that SMART is readily deployable with high portability.

• We evaluate SMART in a real-world server cluster with realistic Hadoop job benchmarks. The experimental results demonstrate that SMART is robust and can effectively reduce the JCT under various job scenarios. Specifically, SMART reduces JCT by up to 6.47%, 9.26%, 13.66% for `Terasort`, `WordCount` and `InvertedIndex` respectively with the PUMA dataset. Furthermore, simulation results show that SMART can reduce JCT by up to 33% as the skewness increases.

2 Background

In this section, we briefly introduce Hadoop MapReduce, Hadoop Yarn and the common methods for task scheduling. In addition, some examples are given to demonstrate the phenomenon of data skew.

2.1 Hadoop MapReduce

MapReduce, popularized by Google [7] and Hadoop①, is a programming framework for distributed

①Hadoop: Developed by Apache Software Foundation. https://hadoop.apache.org, June 2022.

computing programs and the core framework for users to develop "Hadoop-based data analysis applications".

A MapReduce job usually splits the input dataset into independent data blocks, which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the map tasks, which are then inputted to the reduce tasks. Usually, the input and the output of the job are stored in the Hadoop distributed file system (HDFS). The whole framework is responsible for the scheduling and monitoring of tasks, as well as the re-execution of failed tasks.

2.2 Hadoop Yarn

After Hadoop version 2.0, Yarn[②] has been used for scheduling. Yarn is a resource scheduling platform, responsible for providing computing resources for computing programs, which is equivalent to a distributed operating system platform, and computing programs such as MapReduce should be run in applications on top of the operating system.

Yarn is mainly composed of the resource manager (RM), node manager (NM), application master (AM), and containers. The fundamental idea of Yarn is to split up the functionalities of resource management and job scheduling/monitoring into separate modules. In order to do this, the idea is to have a global ResourceManager and per-application ApplicationMaster.

Fig. 2 shows the architecture of Yarn. The functions of components in Yarn are as follows: RM keeps track of NM and available resources, allocates available resources to applications and tasks, and monitors AM. NM is responsible for resource management and task management of a single node. It is responsible for processing requests from RM, and processing container start or stop requests from AM. After receiving the request to start the container, the resources used by the process will be monitored and fed back to RM. AM is responsible for coordinating the execution of all tasks of the current application, and will request containers from RM to run tasks. Container is the resource abstraction within Yarn, describing the running resources of a task (such as memory, CPU, disk and network).

The workflow of Yarn is as follows: the client submits an application to Yarn, and RM allocates a container for the application, communicates with the corresponding NM, and requires NM to start the AM in the allocated container. AM registers with RM so that users can view the running state of the application directly through RM. AM requests resources from RM for each task, and communicates with NM to start the task. After the task has been started, each task reports its state and progresses to its AM. After the application is finished, AM requests RM to log off itself.

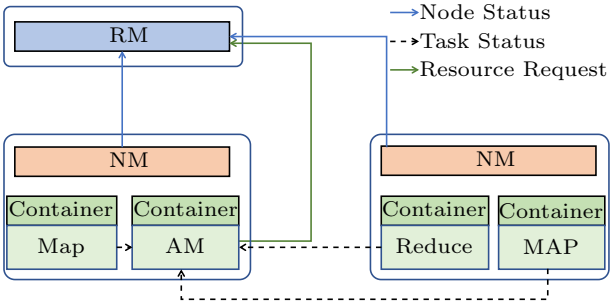


Fig. 2. Architecture of Yarn, which consists of several components, such as RM, NM, AM, and container. NM sends the node state to RM periodically through heartbeat. AM requests resources from RM to run the tasks. Tasks periodically report their current progress and states to AM via heartbeat.

2.3 Task Scheduling

MapReduce jobs can be subdivided into map tasks and reduce tasks. The MapReduce application master divides (MRAppMaster) map tasks and reduce tasks into four states: pending (the resource request has started, but not been sent to RM), scheduled (the resource request has been sent to RM, but RM has not allocated resource for it), assigned (the resource request has been assigned to a container), and completed (the resource request assigned to the container has completed).

The lifecycle of map tasks is: scheduled → assigned → completed. And the lifecycle of reduce tasks is: pending → scheduled → assigned → completed. As is known to all, the execution of reduce tasks depends on the output data of map tasks. In order to avoid the low resource utilization caused by the early startup of reduce tasks, MRAppMaster puts the newly started reduce task in the pending state, so that it can decide whether to schedule it according to the running states of map tasks. Next, we will describe the scheduling of map tasks and reduce tasks in Hadoop respectively.

• *Map Task Scheduling.* When scheduling a map task for a job, the scheduling order is as follows. First, we check if there are any map tasks that have failed to execute. If so, let the failed tasks run first (data locality is not considered here). Second, we assign map tasks

[②]Hadoop Yarn: Developed by Apache Software Foundation. https://hadoop.apache.org/docs/current/hadoop-yarn, June 2022.

which have never been assigned, and select the map task that has the input data closest to the assignable container based on the location first (data location priority is node-local, rack-local, and other-local).

• *Reduce Task Scheduling.* Unlike a map task, reduce tasks do not need to consider data locality because they need to fetch input data from all map tasks. AM starts reduce tasks when the number of completed map tasks reaches a threshold specified by users. In the process of reduce task scheduling, map tasks will be given priority. If map tasks that have applied for resources are not allocated with resources, and the waiting time is too long, the reduce tasks will be preempted, and will not get the opportunity to start and even be killed. If no map task needs to preempt the resources of reduce tasks, reduce tasks will have an opportunity to get scheduled. When several reduce tasks apply for resources concurrently, the resources are allocated in a first-come-first-served manner.

2.4 Phenomenon of Data Skew

As can be seen from the simplified example in Fig.1, when the data is skewed, it is helpful to follow the Largest-First mechanism to schedule reduce tasks.

`Terasort`, `WordCount` and `InvertedIndex` are typical baseline MapReduce tasks. The datasets and programs are publicly available[8]. We take the datasets in `Terasort` (149 GB), `WordCount` (50 GB) and `InvertedIndex` (50 GB) as an example to illustrate the phenomenon of data skew. The output data of all map tasks grouped by partition is accumulated in the MapReduce job, which will be the input data that the reduce tasks need to process.

The results are presented in Fig.3, where the Y-axis denotes the input data size of reduce tasks and the X-axis stands for task IDs. Fig.3 clearly illustrates that the size of data processed by reduce tasks is biased, irrespective of the data scale or job scenarios. This result shows that data skew in reduce tasks is a very common phenomenon.

3 Design

This section gives a formalization of the reduce task scheduling as an optimization problem. And the default FIFO scheduling is discussed. After that we propose the design of SMART, which breaks the scheduling optimization problem into two sub-problems. We first describe how SMART predicts the size relationship among reduce tasks. Then we describe and analyze the largest-first scheduling mechanism based on the prediction.

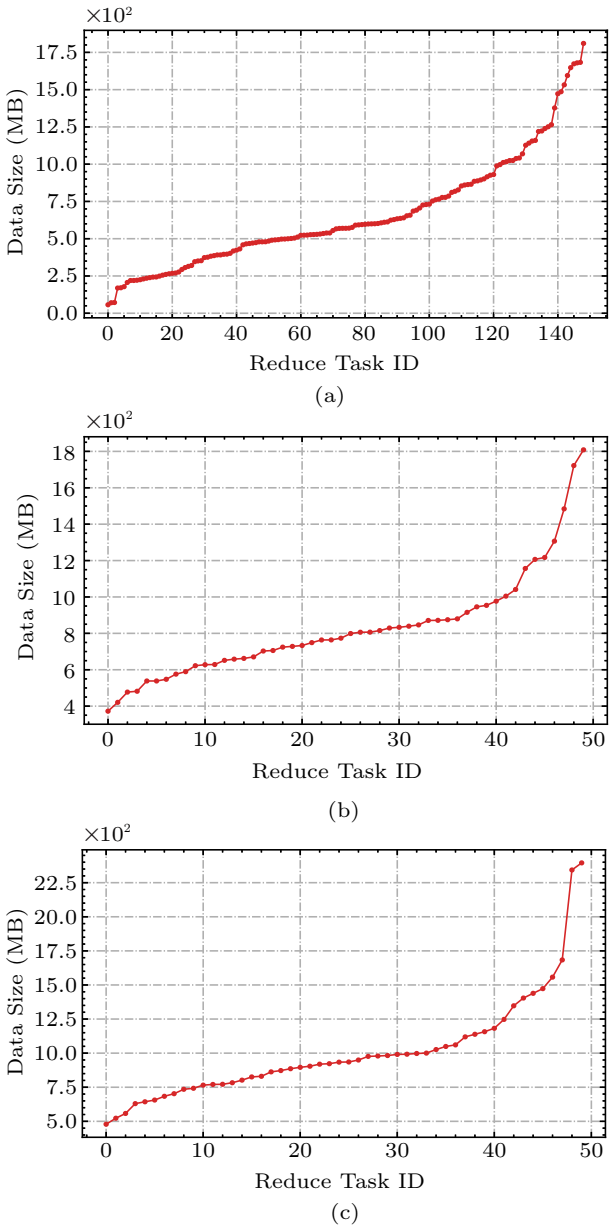


Fig. 3. Data skew of reduce tasks in different scenarios. (a) `Terasort` (149 GB). (b) `WordCount` (50 GB). (c) `InvertedIndex` (50 GB).

3.1 Problem Definition

Multi-job distributed computing systems usually set the threshold of the completed map task for starting the reducing phase to 1, which means the reducing phase is started only after all map tasks have been completed.

Suppose we have m identical containers for the reducing phase and n reduce tasks in total. The i-th reduce task has the task size p_i and takes time t_i to

process. For a specific task scheduling S, let S_j denote the set of tasks scheduled to container j. Then the total working time of container j will be $T_j = \sum_{i \in S_j} t_i$. The target of the job scheduling is to minimize the job completion time of all reduce tasks, which can be defined as:

$$JCT_S = max_j(T_j), \tag{1}$$

where JCT_S stands for the job completion time of the task scheduling S. $max_j(T_j)$ means that the overall job completion time of a task is decided by the maximum total working time of the containers.

It is a reasonable assumption that the processing time required for a reduce task is proportional to its task size, in that all the reduce tasks belonging to the same job execute the same binary code and the processing units in the distributed computing framework usually have the same capability if their hardware devices are the same. In this scenario, the reduce task scheduling problem can be abstracted as a minimum makespan scheduling problem, which is NP-hard.

The minimum makespan scheduling problem can be described as follows. There are m identical machines for processing n jobs each of which takes time t_i to finish. Jobs cannot be split between machines. For a given scheduling, let A_j be the set of jobs assigned to machine j. Let

$$T_j = \sum_{i \in A_j} t_i \tag{2}$$

be the total processing time of machine j.

The target of the minimum makespan scheduling is to find an assignment of jobs to machines that minimizes the makespan, defined as the maximum total processing time of all machines:

$$C_A = max_j(T_j), \tag{3}$$

which is formally consistent with (1), indicating that the job completion time in the reduce task scheduling corresponds to the makespan in the minimum makespan scheduling.

The scheduling logic of the default FIFO is that whenever a container becomes available, it will be assigned to execute the reduce task on top of the queue where tasks are ordered by timestamp. A container becomes available at a specific time t indicates that this container gets the resource to execute a reduce task at time t.

Consequently, the default FIFO mechanism can be described as Algorithm 1, which is consistent with the List [9] algorithm for the minimum makespan problem. Approximation ratio, which is the ratio between the result obtained by an algorithm and the optimal assignment, can be used as a metric to assess algorithms for NP-hard problems. Suppose the makespan of the optimal assignment A^* is C^*, and the result of FIFO scheduling is C_{FIFO}, it can be proved that:

$$\frac{C_{\text{FIFO}}}{C^*} \leqslant 2 - \frac{1}{m}, \tag{4}$$

where m is the total number of machines. (4) indicates that FIFO is a 2-approximation algorithm.

Algorithm 1. Default FIFO Scheduling

Input: $Job_i, i = 1, 2, ..., n$: the task sequence ordered by timestamp
$Container_j, j = 1, 2, ..., m$: all containers for the reducing phase
Output: A_i: the list of (Job_i, $Container_j$) tuples
1: **for** each $Job_i, i = 1, 2, ..., n$ **do**
2: Assign Job_i to a currently available container
3: **end for**

3.2 Design of SMART

In SMART, we break the scheduling optimization problem in the reducing phase into two sub-problems. The first is how to predict the reduce task size. The second is how to schedule the tasks in order to reduce the overall JCT, given the reduce task size.

3.2.1 Task Size Prediction

In Hadoop, it is nontrivial to implement a scheduling mechanism based on the reduce task size because the sizes of reduce tasks are usually unknown a priori. SMART reschedules reduce tasks based on sizes predicted from part of completed map tasks. It is worth noting that SMART does not predict the exact sizes of reduce tasks, but the relative size relationship.

The basic idea of reduce task size prediction in SMART is that the relationship among sizes of reduce tasks is likely to be consistent with the size relationship among intermediate accumulated partitions.

More precisely, let $partition_{\text{total}}[i]$ denote the accumulated partition size for the reduce task i at the time when a fraction p of map tasks have been completed, and let $size_{\text{reduce}}[i]$ denote the size of reduce task i. If $partition_{\text{total}}[m] > partition_{\text{total}}[n]$, then SMART considers that $size_{\text{reduce}}[m]$ will be larger than $size_{\text{reduce}}[n]$. The selection of the value p will be discussed in Subsection 5.2.1. And the correctness of predicted relationship among sizes of reduce tasks will be evaluated in Subsection 5.2.1.

3.2.2 Largest-First Scheduling

With predicted size relationship, SMART is able to schedule reduce tasks with the largest-first mechanism instead of the default FIFO scheduling.

The largest-first scheduling in SMART requires the jobs to be sorted by their task sizes. Whenever a container becomes available, the job with the largest size, which indicates the longest processing time among all rest jobs, will be assigned to that container.

Then the largest-first scheduling can be described as Algorithm 2, which happens to be the LPT (Longest Processing Time) algorithm for the minimum makespan problem. Similarly, it can be proved that this algorithm has the approximation ratio of $4/3 - 1/3m$:

$$\frac{C_{\text{Largest-First}}}{C^*} \leqslant \frac{4}{3} - \frac{1}{3m}, \tag{5}$$

where m is the total number of machines and $C_{\text{Largest-First}}$ stands for the JCT of the largest-first scheduling.

Algorithm 2. Largest-First Scheduling

Input: $Job_i, i = 1, 2, ..., n$: the task sequence
$t_i, i = 1, 2, ..., n$: the processing time of each job
$Container_j, j = 1, 2, ..., m$: all containers for the reducing phase

Output: A_i: list of (Job_i, $Container_j$) tuples

1: Sort $Job[]$ by the task size so that $t_1 \geqslant t_2 \geqslant ... \geqslant t_n$
2: **for** each $Job_i, i = 1, 2, ..., n$ **do**
3: Assign Job_i to a currently least loaded container
4: **end for**

From the analysis above, we can see that the largest-first scheduling is a $\frac{4}{3}$-approximation algorithm for the reduce task scheduling. In contrast, the default FIFO scheduling is a 2-approximation algorithm, which is not so good as the largest-first scheduling. For the worst case of each algorithm, $C_{\text{FIFO}} = 2C^*$, $C_{\text{Largest-First}} = \frac{4}{3}C^*$, and the largest-first scheduling will outperform FIFO by around 33%.

4 Implementation

In this section, we present the detailed implementation of SMART. SMART predicts the size relationship of reduce tasks through obtaining information from RPC messages and then schedules remaining reduce tasks based on the predicted task size relationship.

We describe the architecture of SMART first, then introduce details about how to obtain key information from RPC messages, and finally show how SMART predicts the relative size relationship of reduce tasks.

4.1 Architecture

Fig. 4 illustrates the overall architecture of SMART. SMART resides in a separate node outside the data processing cluster. It continuously monitors the RPC messages in the job to obtain the states of running tasks, collects the output sizes of the completed map tasks, and predicts the sizes of remaining reduce tasks. Note that this process relies on collecting and parsing RPC messages, which does not require any modifications to the application program. It is worth noting that as the algorithms for prediction and scheduling are linear, SMART works with high efficiency. In addition, if there are too many jobs sharing the system, the node where SMART resides can be extended to improve the performance.

In order to get the size of the intermediate data outputted by map tasks, we need to obtain the to-

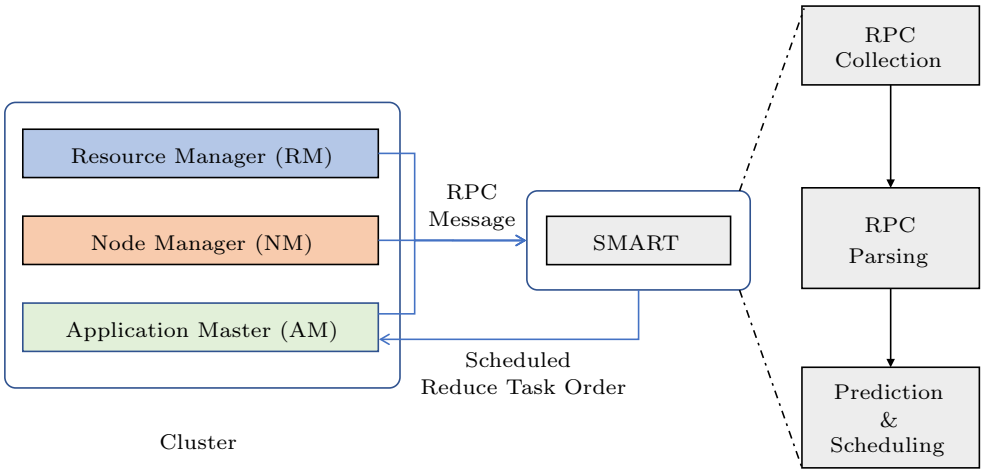


Fig.4. Overall architecture of SMART.

tal number of map tasks and the running states and running positions of each map task. To make minimal changes to the existing system, we develop an RPC collection module (Subsection 4.2) to obtain all RPC messages, and take advantage of the RPC parsing module (Subsection 4.3) to obtain map tasks' information we need. We further develop a prediction module (Subsection 4.4) to predict the reduce task size based on the intermediate results' size from $p = 5\%$ of all map tasks.

4.2 RPC Collection Component

In order to implement this component simply and effectively, we use one dedicated machine to monitor all RPC messages from other machines which compose a cluster running Yarn. We can use a special functionality of switch-port mirroring, which allows the user to send a copy of packets of some ports' traffic stream into an analyzer port[③] to which the specific machine connects. Then we use the open-source library named LIBpcab[④] to sniff and filter packets effectively. All collected messages from the network will be delivered to RPC parsing component for further analysis.

4.3 RPC Parsing Component

Fig. 5 illustrates the transmission mode and function of RPC messages in the cluster, together with how SMART collects different types of information from the system. The following steps present the typical lifecycle of map tasks in a MapReduce job. First of all, the client needs to submit the job to the RM node with function `submitApplication` through ApplicationClientProtocolPB RPC messages. Then the RM node starts the AM node by calling function `startContainers` through sending ContainerManagementProtocolPB RPC messages to the NM node. The AM node will then call function `startContainers` to the NM node to start containers for processing map tasks. Whenever a map task gets completed, the NM node will call the `done` function through TaskUmbilicalProtocol RPC messages. From the above, we can see that transitions between different stages are accompanied by the triggering of RPC messages, from which SMART can parse information about tasks.

RPC messages collected from RPC collection component are binary files. In order to get useful information, SMART deserializes RPC messages first. There are two types of RPC message engines used in Hadoop 3.1, which are ProtobufRpcEngine and WritableRpcEngine. Fig.6 and Fig.7 illustrate the formats of the two types of RPC messages. They correspond to RPC implementation with different serialization/deserialization methods. SMART is able to handle both types of the RPC messages to collect task information.

Algorithm 3 illustrates the details of how SMART parses and filters an RPC message. After intercepting an RPC message, the RpcRequestHeader is used by SMART to determine the type of RPC. SMART uses the protocol name and the method name to filter `submitApplication` and `startContainers` calls from ProtobufRPC messages. In terms of WritableRPC messages, SMART uses the protocol name and the method name to filter `done` calls. All requests matching the filters will be put into specified queues, Q_{sa}, Q_{sc}, Q_{done}, corresponding to three different stages of a map task.

Table 1 lists the kinds of RPC messages SMART needs to parse from the filtered requests.

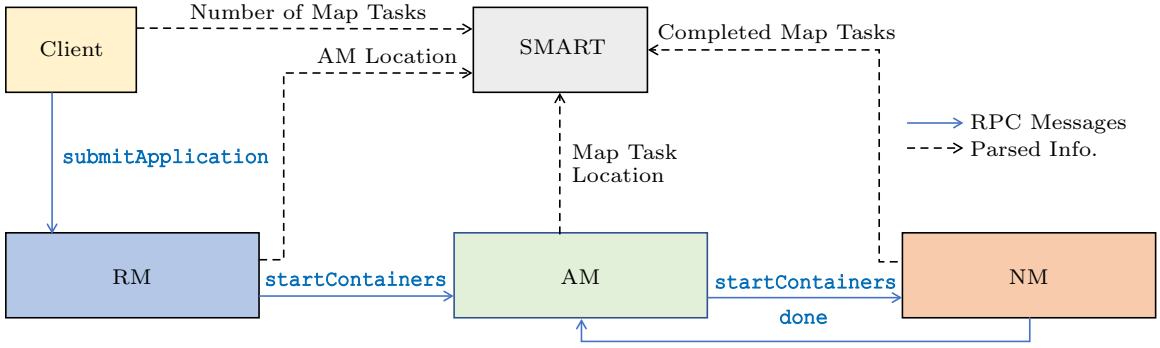


Fig.5. Transmission mode and function of RPC messages in the cluster.

③Mellanox switch onyx-eth-um: By Mellanox technologies. https://www.mellanox.com/files/doc-2020/onyx-eth-um.pdf, June 2022.

④LIBpcap project: By the tcpdump group. https://github.com/the-tcpdump-group/libpcap, June 2022.

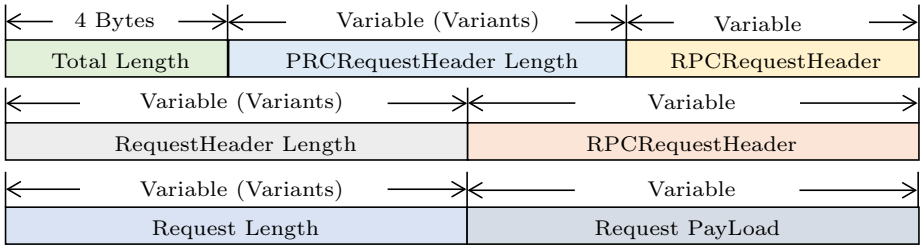


Fig.6. Format of the RPC message using ProtobufRpcEngine for serialization.

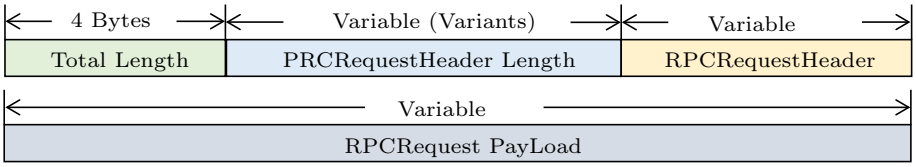


Fig.7. Format of the RPC message using WritableRpcEngine for serialization.

Algorithm 3. Parsing and Filtering the RPC Message

Input: *package*: the RPC message
Q_{sa}: the queue of `submitApplication` requests
Q_{sc}: the queue of `startContainers` requests
Q_{done}: the queue of `done` requests
Output: the target *request* extracted after filtering

```
if RpcRequestHeader.engine == ProtobufRpcEngine then
    if RequestHeader.proto == ApplicationClientProtocolPB && RequestHeader.method == submitApplication then
        Q_sa.push(request)
    else if RequestHeader.proto == ContainerManagementProtocolPB && RequestHeader.method == startContainers then
        Q_sc.push(request)
    else
        delete package
    end if
else if RpcRequestHeader.engine == WritableRpcEngine then
    if RpcRequest.proto == TaskUmbilicalProtocol && RpcRequest.method == done then
        Q_done.push(request)
    else
        delete package
    end if
else
    delete package
end if
```

Table 1. Kinds of RPC Messages Parsed in SMART

RPC Protocol Name	RPC Method Name	RPC Engine	Function
ApplicationClientProtocolPB	`submitApplication`	ProtobufRpcEngine	Get the total number of map tasks from job.xml
ContainerManagementProtocolPB	`startContainers`	ProtobufRpcEngine	Get the location of AM and map tasks
TaskUmbilicalProtocol	`done`	WritableRpcEngine	Indicate that this map task has completed

4.4 Reduce Task Size Prediction Component

From the RPC parsing component, we can get the running location of each map task, the total number of map tasks and the total number of completed map tasks. With the information, SMART is able to obtain the data for prediction when a fraction p of map tasks have been completed. The choice of the value of p will be discussed in Subsection 5.2.1.

Every completed map task has a metadata file named file.out.index, which contains information about its output data. The format of file.out.index is illustrated in Fig. 8, which contains many tuples (start-offset, raw-length, compressed-length). The i-th raw-

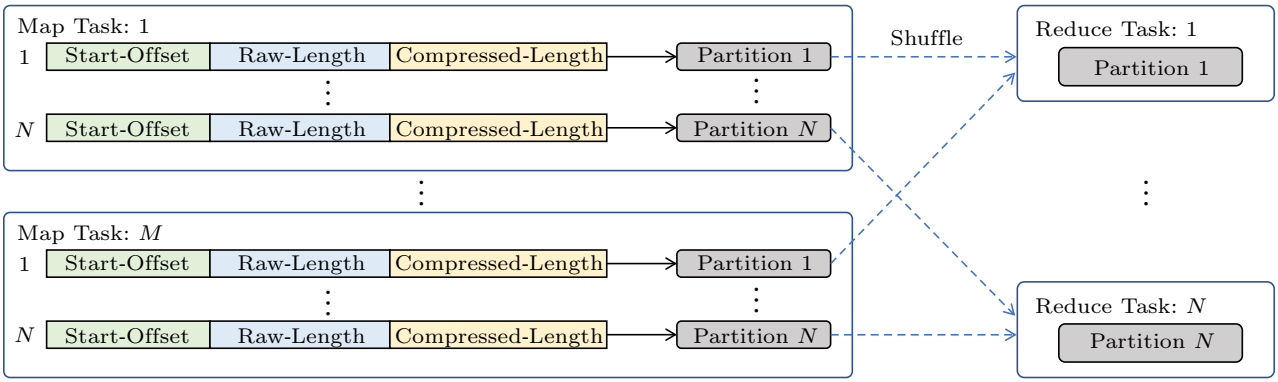


Fig.8. Process of partition data size accumulation.

length represents the i-th partition size of a map task. And, by default, one partition corresponds to one reduce task.

Algorithm 4 describes how SMART extracts the output data size of completed map tasks from file.out.index. SMART keeps monitoring the status of map tasks, tracking their locations and collecting metadata file file.out.index of each completed map task until the number of completed map tasks has reached the threshold ($p = 5\%$). For each completed map task, SMART parses the output data length of different partitions from the metadata file file.out.index. The data length will be the size of data that the reduce task needs to read from the output of the completed map task.

When SMART detects that the number of completed map tasks has reached the threshold ($p = 5\%$), it stops collecting file.out.index. After that, SMART calculates the accumulated partition size of each reduce task from completed map tasks. The detailed algorithm is listed as Algorithm 5. $partition_{\text{total}}[j]$ in Algorithm 5 represents the total length of data in the j-th partition.

By accumulating the size of partitions of completed map tasks, the size of the input data of reduce tasks can be obtained. With the reasonable assumption that the final relative size relationship among reduce tasks will be consistent with the one at the time when a proportion p of total map tasks has been completed, SMART is able to reschedule the reduce tasks based on the predicted relative size relationship. The reorder of reduce tasks will be sent to AM. Finally, AM will schedule reduce tasks with largest-first order.

5 Evaluation

As far as we know, there is no similar work focusing on improving performance of distributed computing systems through rearranging the reducing phase. There is lots of work focusing on reducing JCT from different aspects, such as application-level optimization [10] and data re-partitioning [5, 6, 11]. SMART is orthogonal to

Algorithm 4. Extract the Output Data of map Task from file.out.index

Input: file.out.index for the j-th completed map task
map_num: the total number of map tasks
$job_{\text{Map\_Ratio}}$: the required ratio of completed map tasks
$job_{\text{Has\_Received\_Map}}$: the number of completed map tasks
Output: $map[j].partition[]$: the partition size (output size) list of the j-th map task

1: **repeat**
2: **if** $job_{\text{Has\_Received\_Map}} < (map\_num \times job_{\text{Map\_Ratio}})$ **then**
3: file.out.index is composed of tuples (start-offset, raw-length, compressed length)
4: **for** each $i \in [1, partition\_num]$ **do**
5: Initialize an array $data$ to save tuples as above
6: fread($data$, sizeof($tuple$), file.out.index) // $data[1]$ is raw-length
7: $map[j].partition[i] = data[1]$
8: **end for**
9: **end if**
10: **until** $job_{\text{Has\_Received\_Map}} \geqslant (map\_num \times job_{\text{Map\_Ratio}})$

these existing studies and can bring additional benefit, as long as data skew exists. Therefore the evaluation focuses on assessing the effectiveness of SMART itself.

Algorithm 5. Accumulating the Value of Partitions

Input: the list of partitions sizes of completed map tasks
Output: the order of reduce tasks
1: **for** each $i \in [1, map\_num]$ **do**
2: **for** each $j \in [1, partition\_num]$ **do**
3: Get partition length of the i-th map task with Algorithm 4
4: $partition_{\text{total}}[j]$ += $map[i].partition[j]$
5: **end for**
6: **end for**
7: $reduce\_task\_order = sort(partition_{\text{total}}, desc)$

This section evaluates SMART from several aspects. Firstly, we evaluate the robustness of SMART by making different choices of parameter p. Then the effectiveness of SMART is evaluated against different datasets. Finally, we discuss the influence of different skewness over JCT.

5.1 Methodology

Experimental Setup. SMART is evaluated in a real-world server cluster, which is composed of 13 physical servers. One of the servers is used for RM, one is used for SMART, and the others are used as computing nodes. All the physical servers are NUMA (non-uniform memory access) architecture with Intel® Xeon® Silver 4214 2.2 GHz CPU, two sockets, and each socket contains 12 cores, 256 GB memory, 100 Gb/s network interface. The switch used in the cluster is Mellanox SN2700. We run Hadoop 3.1 on the physical servers for the SMART evaluation. HDFS is stored in memory, where the size of an HDFS block is 128 MB and the replication factor is set to 3. The resources ⟨CPU, memory⟩ assigned to each map task and reduce task are ⟨1 core, 1.5 GB⟩. Same as most of the multi-job systems, the slowstart factor in SMART is set to 1, that is, the reducing phase is started only after all map tasks are completed. The scheduler used by Hadoop is the capacity scheduler. Unless otherwise specified, the experimental setup remains unchanged.

Workload. In order to test different scenarios, we setup three different experiments for the evaluation, i.e., `Terasort`, `WordCount` and `InvertedIndex`.

For `Terasort`, we use `teragen` in hadoop-mapreduce-examples-3.1.1.jar to generate different sizes of the dataset (10 GB, 15 GB, 149 GB, etc.) as input data. We also use the Wikipedia dataset (50 GB, 80 GB, 280 GB) in [8] as the input data for `WordCount` in hadoop-mapreduce-examples-3.1.1.jar and for `InvertedIndex` in puma.jar [8].

The programs and datasets are publicly available on the website [8].

Evaluation Metrics. In the evaluation, we use JCT as the metric. We can obtain JCT from *finishTime* and *submitTime* from the job_id.summary file: $JCT = finishTime - submitTime$.

5.2 Effectiveness of SMART

This subsection evaluates the effectiveness of SMART through experiments with different data scales under different scenarios.

The computing resource ⟨CPU, memory⟩ assigned to each map task and reduce task is ⟨1 core, 2 GB⟩.

5.2.1 Choice of Parameter p

The parameter p in SMART denotes the threshold for the ratio of completed map tasks. SMART waits until the ratio of completed map tasks reaches the threshold and then predicts the size of reduce tasks based on the output of the currently completed map tasks.

The choice of parameter p faces a tradeoff between accuracy and cost. It is straightforward that as p increases, SMART can have more intermediate data to predict the size of reduce tasks and thus will be able to have a more accurate largest-first order of reduce tasks. However, as a large number of map tasks are executed parallelly for one job, the amount of data processed by each map is small, which results in that the execution speed of the map phase is fast. SMART needs to collect and parse all RPC messages of p map tasks in a short time. If p is set too large, it will be too late for SMART to send the rearranged reduce order before AM starts to schedule the reduce tasks.

To evaluate the robustness of SMART against parameter p, we evaluate the effectiveness of SMART under different job scenarios with a large data scale. Specifically, different values of p (0.05, 0.125, 0.25) are evaluated under `Terasort`, `WordCount` and `InvertedIndex` with 267 GB, 280 GB and 280 GB dataset respectively.

Fig.9 is an intuitive task scheduling diagram in the `Terasort` job, with the X-axis representing the running time of containers and the Y-axis representing different container IDs. Each colored bar in Fig.9 represents the running time of a specific reduce task. For example, for a red bar with the Y-axis value 70, the abscissa range

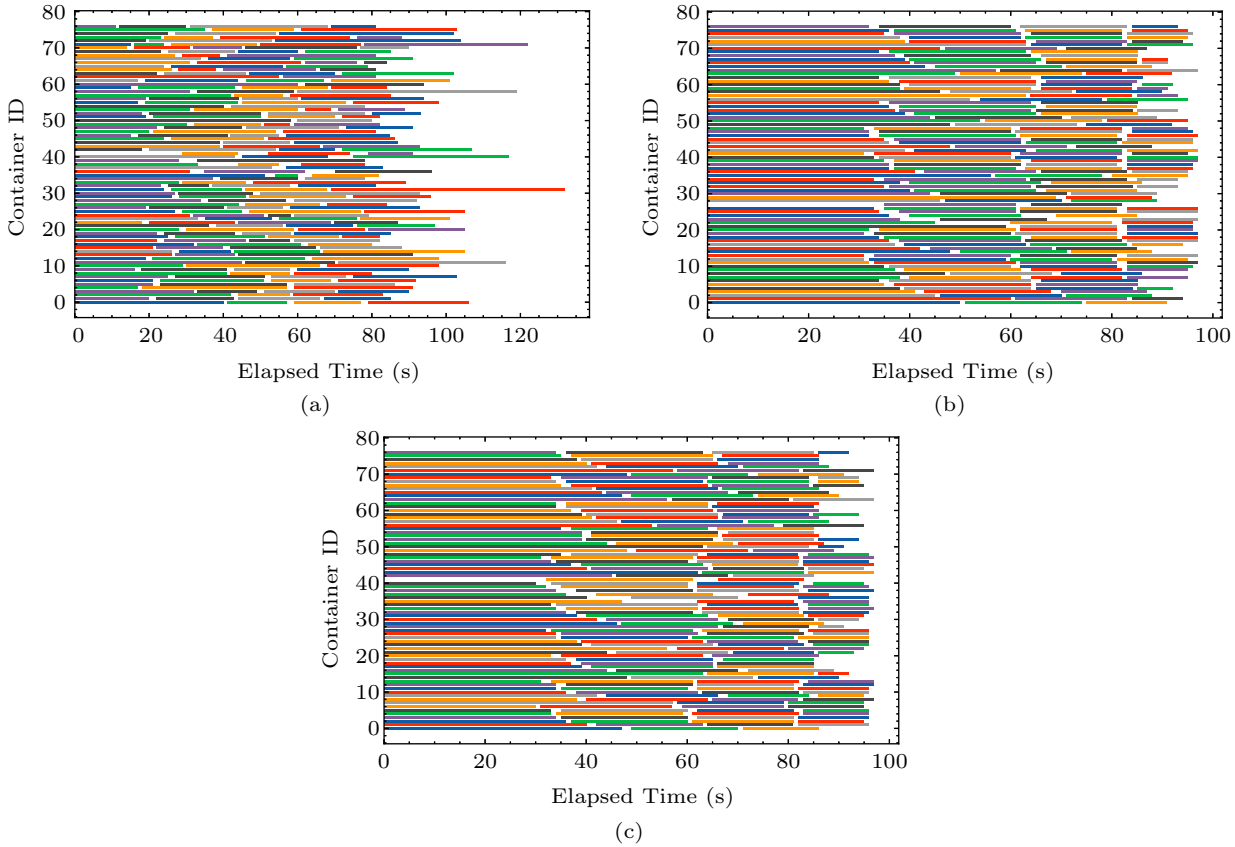


Fig.9. Reduce task scheduling diagram: the reduce task is running in the container. A horizontal bar represents the running time of a particular container. (a) Native Hadoop. (b) SMART with $p = 0.05$. (c) SMART with $p = 0.125$.

of the line is [15, 40], which means that the reduce task is assigned to the container with ID 70 and runs from 15 s to 40 s and the total running time is 25 s. As the baseline, the reduce task scheduling result of the native Hadoop is shown in Fig.9(a), in which reduce tasks are randomly distributed and run loosely. Figs.9(b) and 9(c) are the scheduling diagram of SMART, with parameter p set to 0.05 and 0.125 respectively. The running time of the reducing phase depends on the maximum abscissa value in Fig.9. For instance, the abscissa value of the last bar in Fig.9(a) exceeds 120 while the maximum abscissa value of the last bar in Figs.9(b) and 9(c) does not exceed 100, indicating that the running time of the reducing phase in native Hadoop is longer than the running time in SMART. Compared with the baseline, the result tells that when long reduce tasks are scheduled first, the entire scheduling will be comparatively compact, and the total running time of the reducing phase will also be reduced. It is worth noting that even though the p value in Fig.9(c) is over twice as large as the p value in Fig.9(b), the scheduling results are almost the same.

To better understand the robustness of SMART against parameter p, we define the metric sortedness S:

$$S = 1 - \frac{2}{N^2} \sum_{i=1}^{N} |f(x_i) - i| , \tag{6}$$

where N is the total number of elements in the list, x_i is the i-th element in the sorted list and $f(x_i)$ is the position or index of element x_i in the unsorted list that we want to measure. The metric S ranges from 0, for completely unsorted, to 1, for completely sorted.

Fig.10 shows sortedness of SMART with different values of parameter p under different job scenarios. From the result we can see that with a small p (0.05), the sortedness of scheduled reduce tasks can be improved notably, while the marginal benefit brought by larger p is negligible. The appropriate value of p mainly depends on data skewness. When the data skewness is large, the predicted sorting order can be obtained with fewer map tasks (smaller p). For the evaluation in this paper, it can be seen from Fig.3 that the data in these experiments is very skewed. Therefore, as illustrated in Fig.10, when the values of p are 0.05, 0.125 and 0.250

respectively, the sortedness is almost the same. With a small p (0.05), the sortedness of scheduled reduce tasks can be improved notably.

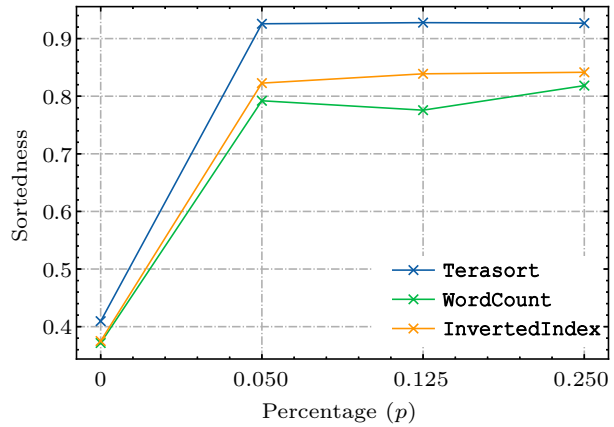


Fig.10. Sortedness with different values of parameter p under different job scenarios.

Fig.11 shows the JCT reduction ratio of `Terasort` 80 GB/267 GB with parameter p set to 0.05, 0.125, 0.250 respectively. It can be seen that the performance improvement under different values of p (0.05, 0.125, 0.250) is very similar.

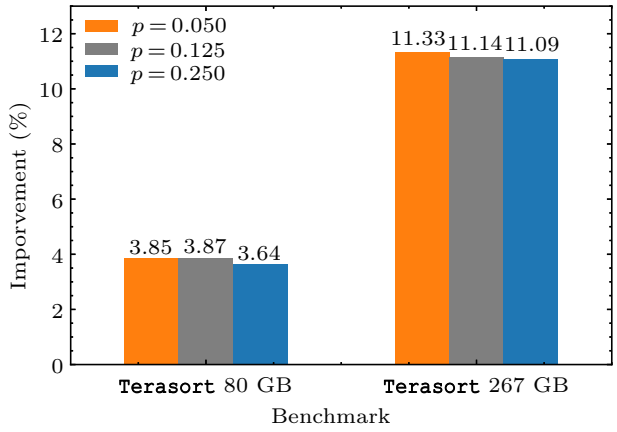


Fig.11. Improvement of `Terasort` under different p values and different sizes.

The above results demonstrate that SMART is robust against parameter p and it is sufficient to choose the value parameter p as 0.05.

5.2.2 *Effectiveness Under Different Scenarios*

In this subsection, we evaluate performance of SMART under different job scenarios.

Table 2 records the workload of different job scenarios in this evaluation. Three different job scenarios are tested, each with two datasets of different data scales. The parameter p is set to 0.05 as guided by Subsection 5.2.1.

Table 2. Workload of Different Job Scenarios

Scenario	Input Data	Data Size
`Terasort`	Generated by `teragen`	80 GB, 280 GB
`WordCount`	Wikipedia	80 GB, 280 GB
`InvertedIndex`	Wikipedia	80 GB, 280 GB

The experimental results are presented in Fig. 12. From the results we see that SMART can effectively improve the performance of different jobs. In addition, Fig.12 also tells that SMART performs better as the input size increases. This is because SMART is able to gain more scheduling space as the data size increases. However, the input data size is not the most critical factor, a more important factor is the skewness of the input data.

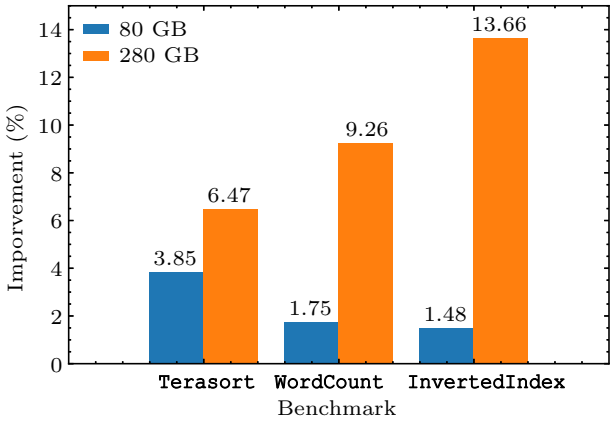


Fig.12. Performance improvement under different job scenarios.

5.2.3 *Skewness Simulation*

We make a simulation to see the performance improvement brought by SMART with the strict implementation of largest-first scheduling when the data is skewed. We define a metric skewness K:

$$K = \frac{Size_{+10\text{th}}}{Size_{-10\text{th}}}, \tag{7}$$

where $Size_{+10\text{th}}$ is the average value of the largest 10% reduce task size, and $Size_{-10\text{th}}$ is the average value of the top 10% smallest reduce task size.

We generate datasets with different skewness as the input data and evaluate the effectiveness of SMART.

The experimental results are presented in Fig. 13. It can be seen from Fig.13 that performance improvement brought by SMART increases as the skewness of datasets increases, and finally stabilizes at a performance improvement ratio around 33%, which accords with the theoretical analysis in Subsection 3.2.2.

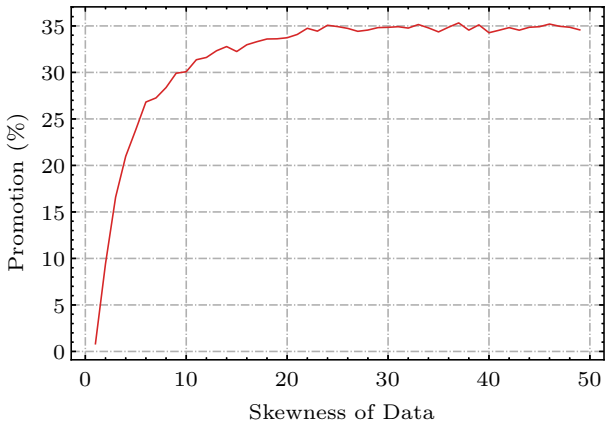


Fig.13. Performance improvement under different data skewness.

6 Related Work

There have been lots of researches focusing on improving resource utilization in distributed computing frameworks to reduce JCT in recent years. In order to achieve the goal, they try to optimize distributed systems from different aspects, including but not limited to, resource utilization and job scheduling.

In the aspect of network, Mosharaf and Ion [10] focused on application-level networking abstraction and proposed that coflow scheduling [12, 13] can achieve high network utilization and decrease communication time.

In the aspect of directed acyclic graph (DAG), Grandl *et al.* [1] suggested that scheduling heterogeneous DAG can not only improve the throughput of cluster, but also reduce JCT. Mao *et al.* [14] used reinforcement learning and past workload logs to learn sophisticated scheduling policies automatically, which can achieve efficient resource utilization and reduce JCT.

In the aspect of memory usage, Nguyen *et al.* [15, 16] observed that big data applications' memory usage is different with other applications, such that optimizing memory management policy in Java virtual machine (JVM) can reduce garbage collection time, thus reducing JCT.

In the aspect of disk I/O, Rasmussen *et al.* [17] improved the performance of MapReduce jobs by ensuring the intermediate data not being repetitively accessed through disks, which can reduce the amount of random disk I/O access. Rao *et al.* [18] used a new file system design to support multiple insertion points to aggregate intermediate results, thus reducing disk access time. Zhang *et al.* [19] observed that there are a large number of small I/O requests during the shuffle stage of large jobs, which can incur significant shuffle overhead, and suggested that merging fragmented intermediate files can convert small, random disk IO requests to large, sequential ones, thus reducing completion time.

In addition, MapReduce jobs can be scheduled at the lower granularity of tasks. MapReduce jobs consist of two separate phases: the mapping phase and the reducing phase, which are scheduled independently. Therefore, many papers take map scheduling and reduce scheduling as research points.

• *Map Task Scheduling.* Map task scheduling mainly focuses on data locality. Zaharia *et al.* [20] proposed delay scheduling. It schedules a task to run on some hosts which contain the task's input data. It reduces JCT by temporarily sacrificing fairness to satisfy data locality. Ibrahim *et al.* [21] proposed the Maestro algorithm to schedule map tasks by tracking data blocks and copy locations to ensure data locality. The above two solutions use data locality to reduce network transmission to achieve performance improvement. However, in a high-speed network environment, the network cannot be fully used and the proportion of network transmission is relatively small, and therefore the utilization of data locality cannot be better improved.

• *Reduce Task Scheduling.* Hammoud and Sakr [6] proposed Lars, trying to put the reduce task on the node with the largest input data to ensure data locality. As described above, the utilization of data locality cannot be better improved. Tang *et al.* [22] proposed to reduce JCT by dynamically determining the start time of the reduce task. However in the high-speed network environment, the performance improvement by reducing the data copy time is small. In addition, data skew may occur in the reducing phase. Data skew in Hadoop is an imbalance in the load assigned to different reduce tasks. The load includes the number of keys assigned to a reducer, the number of values, and the number of bytes of each value. When faced with the problem of data skew in the reducing phase, the data can be re-partitioned to achieve balance [5, 6, 11], thus reducing JCT. Kwon *et al.* [11] proposed an optimizer SkewTune, parameterized by user-defined cost functions. Through user-defined cost functions, SkewTune determines how best to partition the input data to minimize computational skew. IbrahIm *et al.* [23] developed LEEN, an algorithm for locality-aware and fairness-aware key partitioning in MapReduce. The intermediate keys after the shuffle phase are partitioned according to their frequencies and the fairness of the expected data distribution.

• *Coupling Task Scheduling.* Tan *et al.* [24] proposed a resource-aware scheduler, which couples the

progresses of map tasks and reduce tasks to optimize task placement for both of them. It improves the overall data locality; thus it improves the job response time.

7 Conclusions

In this work, we presented SMART, a readily deployable module for Hadoop that can effectively reduce the JCT through handling the data skew during the reducing phase. SMART predicts the reduce task size based on part of the completed map tasks and then enforces largest-first scheduling in the reducing phase according to the predicted reduce task size. SMART proves to be robust and effective. The experimental results showed that SMART reduces the job completion time by up to 6.47%, 9.26%, 13.66% for `Terasort`, `WordCount` and `InvertedIndex` respectively with 280 GB dataset compared with the native Hadoop.

In the future work, we consider optimizing the prediction of the reduce task size through adding an offline training module. This paper takes Hadoop as a specific case; however, SMART is not Hadoop-specific. Theoretically, it can be applied to data processing systems with DAG patterns, under the assumption that processing units are limited and the input of some tasks are the output of some other tasks. MapReduce jobs in Hadoop can be regarded as a specific kind of DAG-task, where the input of reduce tasks is the output of map tasks. We plan to implement SMART in other distributed frameworks such as Spark in the future.

Acknowledgement The authors would like to thank the anonymous reviewers for their valuable feedback.

References

[1] Grandl R, Kandula S, Rao S, Akella A, Kulkarni J. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*, Nov. 2016, pp.81-97.

[2] Chang H, Kodialam M, Kompella R R, Lakshman T V, Lee M, Mukherjee S. Scheduling in MapReduce-like systems for fast completion time. In *Proc. the 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies*, Apr. 2011, pp.3074-3082. DOI: 10.1109/INFCOM.2011.5935152.

[3] Peng Y, Chen K, Wang G, Bai W, Zhao Y, Wang H, Geng Y, Ma Z, Gu L. Towards comprehensive traffic forecasting in cloud computing: Design and application. *IEEE/ACM Transactions on Networking*, 2016, 24(4): 2210-2222. DOI: 10.1109/TNET.2015.2458892.

[4] Ullah I, Khan M S, Amir M, Kim J, Kim S M. LSTPD: Least slack time-based preemptive deadline constraint scheduler for Hadoop clusters. *IEEE Access*, 2020, 8: 111751-111762. DOI: 10.1109/ACCESS.2020.3002565.

[5] Gao Y, Zhou Y, Zhou B, Shi L, Zhang J. Handling data skew in MapReduce cluster by using partition tuning. *Journal of Healthcare Engineering*, 2017, 2017: Article No. 1425102. DOI: 10.1155/2017/1425102.

[6] Hammoud M, Sakr M F. Locality-aware reduce task scheduling for MapReduce. In *Proc. the 3rd IEEE International Conference on Cloud Computing Technology and Science*, Nov. 29-Dec. 1, 2011, pp.570-576. DOI: 10.1109/CloudCom.2011.87.

[7] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 2008, 51(1): 107-113. DOI: 10.1145/1327452.1327492.

[8] Ahmad F, Lee S, Thottethodi M, Vijaykumar T. PUMA: Purdue MapReduce benchmarks suite. Technical Report, Purdue University, 2012. https://engineering.purdue.edu/∼puma/puma.pdf, May 2022.

[9] Graham R L. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 1966, 45(9): 1563-1581. DOI: 10.1002/j.1538-7305.1966.tb01709.x.

[10] Mosharaf C, Ion S. Coflow: A networking abstraction for cluster applications. In *Proc. the 11th ACM Workshop on Hot Topics in Networks*, Oct. 2012, pp.31-36. DOI: 10.1145/2390231.2390237.

[11] Kwon Y, Balazinska M, Howe B, Rolia J. SkewTune: Mitigating skew in MapReduce applications. In *Proc. the 2012 ACM SIGMOD International Conference on Management of Data*, May 2012, pp.25-36. DOI: 10.1145/2213836.2213840.

[12] Chowdhury M, Zhong Y, Stoica I. Efficient coflow scheduling with Varys. *ACM SIGCOMM Comput. Commun. Rev.*, 2014, 44(4): 443-454. DOI: 10.1145/2740070.2626315.

[13] Chowdhury M, Stoica I. Efficient coflow scheduling without prior knowledge. In *Proc. the 2015 ACM Conference on Special Interest Group on Data Communication*, Aug. 2015, pp.393-406. DOI: 10.1145/2785956.2787480.

[14] Mao H, Schwarzkopf M, Venkatakrishnan S B, Meng Z, Alizadeh M. Learning scheduling algorithms for data processing clusters. In *Proc. the ACM Special Interest Group on Data Communication*, Aug. 2019, pp.270-288. DOI: 10.1145/3341302.3342080.

[15] Nguyen K, Wang K, Bu Y, Fang L, Hu J, Xu G. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *Proc. the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2015, pp.675-690. DOI: 10.1145/2694344.2694345.

[16] Nguyen K, Fang L, Xu G, Demsky B, Lu S, Alamian S, Mutlu O. Yak: A high-performance big-data-friendly garbage collector. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation*, November 2016, pp.349-365.

[17] Rasmussen A, Lam V T, Conley M, Porter G, Kapoor R, Vahdat A. Themis: An I/O-efficient MapReduce. In *Proc. the 3rd ACM Symposium on Cloud Computing*, Oct. 2012, Article No. 13. DOI: 10.1145/2391229.2391242.

[18] Rao S, Ramakrishnan R, Silberstein A, Ovsiannikov M, Reeves D. Sailfish: A framework for large scale data processing. In *Proc. the 3rd ACM Symposium on Cloud Computing*, Oct. 2012, Article No. 4. DOI: 10.1145/2391229.2391233.

[19] Zhang H, Cho B, Seyfe E, Ching A, Freedman M J. Riffle: Optimized shuffle service for large-scale data analytics. In *Proc. the 13th EuroSys Conference*, Apr. 2018, Article No. 43. DOI: 10.1145/3190508.3190534.

[20] Zaharia M, Borthakur D, Sarma S J, Elmeleegy K, Shenker S, Stoica I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. the 5th European Conference on Computer Systems*, Apr. 2010, pp.265-278. DOI: 10.1145/1755913.1755940.

[21] Ibrahim S, Jin H, Lu L, He B, Antoniu G, Wu S. Maestro: Replica-aware map scheduling for MapReduce. In *Proc. the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2012, pp.435-442. DOI: 10.1109/CCGrid.2012.122.

[22] Tang Z, Jiang L, Zhou J, Li K, Li K. A self-adaptive scheduling algorithm for reduce start time. *Future Generation Computer Systems*, 2015, 43/44: 51-60. DOI: 10.1016/j.future.2014.08.011.

[23] Ibrahim S, Jin H, Lu L, Wu S, He B, Qi L. LEEN: Locality/fairness-aware key partitioning for MapReduce in the cloud. In *Proc. the 2nd IEEE International Conference on Cloud Computing Technology and Science*, Nov. 30-Dec. 3, 2010, pp.17-24. DOI: 10.1109/CloudCom.2010.25.

[24] Tan J, Meng X, Zhang L. Coupling task progress for MapReduce resource-aware scheduling. In *Proc. the 2013 IEEE INFOCOM*, Apr. 2013, pp.1618-1626. DOI: 10.1109/INFCOM.2013.6566958.

Jia-Qing Dong received his Ph.D. degree in computer science and technology from Tsinghua University, Beijing, in July 2020 and received his B.S. degree in computer science and technology from Peking University, Beijing, in July 2013. He is currently an assistant researcher with the State Key Laboratory of Media Convergence and Communication, Communication University of China, Beijing. His research interests include data center networks, and distributed systems.

Ze-Hao He received his M.S. degree in computer science and technology from Nanjing University, Nanjing, in 2021, and his B.S. degree in computer science and technology from Shandong University, Jinan, in 2018. His research interests include datacenter networks and systems.

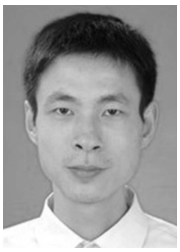
Yuan-Yuan Gong received her B.S. degree in software engineering from Hunan University, Changsha, in 2019, and her M.S. degree in computer science and technology from Nanjing University, Nanjing, in 2022. Her research interests include the task scheduling of big data system.

Pei-Wen Yu received his B.E. degree in environment engineering at Nanjing University, Nanjing, in 2020. He is a M.E. student in the Department of Computer Science and Technology at Nanjing University, Nanjing. His research interests include datacenter networks and network architecture.

Chen Tian received his B.S., M.S., and Ph.D. degrees in communication engineering from Huazhong University of Science and Technology, Wuhan, in 2000, 2003, and 2008, respectively. He is currently a professor with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing. From 2012 to 2013, he was a postdoctoral researcher with the Department of Computer Science, Yale University, New Haven. His current research interests include data center networks, network function virtualization, distributed systems, and Internet streaming.

Wan-Chun Dou received his Ph.D. degree in mechanical and electronic engineering from the Nanjing University of Science and Technology, Nanjing, in 2001. He is currently a full professor of the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing. He visited the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, as a visiting scholar in 2005 and 2008. His research interests include workflow, cloud computing, and service computing.

Gui-Hai Chen received his B.S. degree in computer software from Nanjing University, Nanjing, in 1984, his M.E. degree in computer applications from Southeast University, Nanjing, in 1987, and his Ph.D. degree in computer science from the University of Hong Kong, Hong Kong, in 1997. He is a distinguished professor of Nanjing University, Nanjing. He had been invited as a visiting professor by the Kyushu Institute of Technology, Kitakyushu, University of Queensland, St Lucia, and Wayne State University, Detroit. He has a wide range of research interests with focus on parallel computing, wireless networks, data centers, peer-to-peer computing, high-performance computer architecture, and data engineering.



Nai Xia received his B.S. (2001), Ph.D. (2007) degrees in computer science and technology from Nanjing University, Nanjing. He is an assistant professor with Department of Computer Science and Technology, Nanjing University, Nanjing. His research interests focus on operating system design and implementation.

Hao-Ran Guan is currently pursuing his B.S. degree in data science and software development in School of Computer Science, University of Sydney, Sydney. His research interests include data analysis and machine learning systems.