MR. Prophet: Predicting the Performance of MapReduce Applications by Considering Phase Parallelism

Zhonghou Lv†,Hailong Sun‡,Xu Wang†,Xudong Liu†
School of Computer Science and Engineering
Beihang University
Beijing, China
†{*lvzh, wangxu, liuxd*}*@act.buaa.edu.cn,* ‡*sunhl@buaa.edu.cn*

***Abstract*—MapReduce has been proved to be a successful programming model for big data processing. An accurate performance model has significant implication for optimizing the performance of MapReduce applications, which highly relies on the accurate anatomy of the execution process of a MapReduce application. To speedup the processing of a MapReduce job, parallelism between multiple phases inside the task is widely employed in Hadoop and this parallelism is closely related to a set of configuration parameters at the same time. Surprisingly we find that existing works on the MapReduce performance model scarcely consider the multi-phase parallelism, and most of efforts simply divide the task into a sequential flow, such as read, map, spill and merge inside the map task. In this work, we propose a performance model for MapReduce applications, MR. Prophet, by fully incorporating the multi-phase parallelism inside the task into the model. We also design and implement a lightweight performance instrumentation tool, LTrace, to collect some application and system specific statistical data so as to instantiate the model for an application. Finally, we have conducted a set of experimental evaluation of our model with HiBench workload. The results illustrate that MR. Prophet can achieve very high prediction accuracy and greatly outperforms the comparing model, for example, our model can achieve above 94% accuracy for WordCount, TeraSort and PageRank jobs.**

***Keywords*-MapReduce; performance model; parallelism;**

I. Introduction

In recent years, with the rapid development of e-commerce, search engines, social networks and mobile internet, the challenge of data processing has been growing rapidly due to the fast-growing volumes of data. MapReduce [1] is one of the most popular and efficient big data computing framework. Hadoop [2] is the most frequently-used open source implementation of MapReduce. Programmers, researchers and even some computer science beginners can deal with big data processing jobs, such as log analysis, index building and data mining, on the platform of hadoop.

A MapReduce Job consists of three parts: user-defined programs, input data and configuration parameters. Configuration parameters are the user-specified set of options. The selection of configuration parameters has a significant impact on the execution overhead of a MapReduce job, and the user can optimize the parameters to speedup job processing [3], [4], [5]. However, this is a great challenge for the IT professionals, even for people who have a good understanding of Hadoop's internal mechanism, to select the most effective set of parameters. To optimize the configuration parameters more effectively, a productive tool is necessary for users to predict MapReduce job's execution time with varying configuration parameters.

Nowadays, there are some related research [3], [4], [6] on the prediction of MapReduce job's execution time, which can be applied to optimize the configuration parameters [7]. The construction of job's performance model is one of the most effective methods to forecast the execution time of MapReduce jobs. However, there are some deficiencies in the existing performance models. The present models are constructed to predict the executive cost of each individual phase of MapReduce, and the simple addition of each phase's cost is the prediction of the execution time. As a matter of fact, the executing procedure of a MapReduce job in Hadoop system is complicated, which is not computed step by step purely. For example, multi-threading technology leads to the parallel execution between multiple individual phases.

To construct an accurate performance model for the MapReduce jobs, it is necessary to confirm whether the parallelism exists between multiple phases, and to measure the job's execution time in the presence of parallelism. We have analyzed the practical execution process of Hadoop MapReduce through reading its source code. As a result, we find that some fine-grained phases' execution parallels with others'. Furthermore, we have conducted a thorough analysis of the associated factors that affect the execution time of each fine-grained phase and the overlap time between multiple phases.

In addition, we need an extra lightweight monitoring and analysis tool to obtain the execution statistical data of MapReduce Jobs, and these statistics contain the statistical information about the dataflow and execution time of each fine-grained phase. The recent related works use the third-party profilers to capture the executive statistics, and this profiler increases the overhead by 5%-30%, which negatively affects the execution performance of the running MapReduce

Jobs and consequently brings errors to the accuracy of statistics. To relieve the negative impact of the profiler's heavy overhead, we design a lightweight approach, which is based on the log printing system, to collect the statistics. The row data of the running jobs' execution information is printed to the log files through the log message firstly, and then the statistics are obtained off-line through the extraction of the log files. This approach separates the statistics collection into two phases, which minimize the negative impact of the profiler's overhead.

This paper introduces how to predict the execution time through the construction of the MapReduce performance model, the main contributions of this paper are as follows:

1) we construct a MapReduce performance model MR. Prophet, which can predict the execution time of a MapReduce job with high accuracy.
2) we design a lightweight log-based approach to capture the execution statistics of MapReduce jobs.
3) we have conducted extensive experiments with HiBench, and verified the accuracy of Mr. Prophet through the comparison with the related work.

The rest of this paper is organized as follows. Section 2 presents a fine anatomy of the processing of MapReduce jobs. Section 3 introduces the design and implementation of the lightweight profiler, LTrace. Section 4 describes a new Hadoop MapReduce's performance model. Section 5 presents the evaluation results with HiBench workload. And Section 6 introduces some related work, finally Section 7 concludes this paper.

II. Anatomy of MapReduce Job Execution

This section introduces the executing procedure of running a MapReduce job in Hadoop.

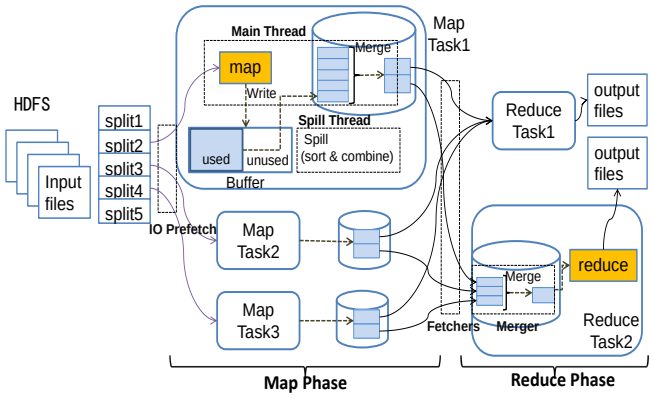

Figure 1. The execution procedure of map and reduce tasks

Map Tasks: When the needed resources are allocated to execute a map task, a java process is started on the corresponding node, then the execution environment is initialized. Next the Java process begins to read the input data and run the user-defined Map function. Due to the use of the pre-fetch I/O, the input data to be processed in the map function is read from the input files while the map function is processed. As a consequence, the use of the pre-fetch I/O effectively reduces the cost of reading the data from input files.

As shown in Figure 1, the execution results of a map function are written into a memory buffer while the size of this buffer is determined by a configuration parameter mapreduce.task.io.mb. A spill thread is started to spill the data of the buffer to the local disks, when the used space of the buffer reaches a pre-configured threshold. As seen above, the procedure of spill consists of two fine-grained phases: *sort* and *combine*. The spill thread first sorts the data of the buffer, then combines and writes the data into a new local disk file. At the beginning of spill, as there is enough buffer space available, the main thread of this java process can still execute the map function and write the output into the buffer until the buffer is full. Therefore, the spill thread, which spills the data of the buffer to the disk, runs in parallel with the main thread to some extent. After the last spill, the main thread merges the outputs of the multiple spills into a new file in some way.

Reduce Tasks: After some of the map tasks have finished, a new java process is launched to execute reduce tasks. Reduce tasks are initialized first, this java process starts to retrieve data from the output of map tasks. To that end, multiple fetch threads are started to *copy* the outputs of the completed map tasks from remote machines simultaneously, where the number of threads is also a pre-configured parameter. A fetch thread sequentially reads the output of each map task, and the retrieved data is written to the memory or the disk, which is determined by the size of output. In general, the use of multiple fetch threads can accelerate the data shuffle process.

At the same time, a *merge* thread is started to merge the retrieved data either in the memory buffer or on the disk. The merge is run when the used buffer space or the number of disk files reaches the respective pre-configured thresholds. And in this process, the fetch threads and the merge thread normally run in parallel. After all the data has been fetched and merged, the reduce function will be executed. Finally, the results of reduce function will be written to HDFS[8] as the job processing results.

III. A Lightweight Profiler

We design and implement a lightweight profiler, which is called LTrace, to capture the execution statistics. The design of the LTrace is introduced in the following content.

Design Goals: The new profiler to capture the execution statistics should address the following characteristics.

- **Lightweight:** The execution of the profiler consumes some resources of the cluster, which affects the execution performance of the running MapReduce jobs. In order to minimize the impact on the running MapReduce jobs, the profiler LTrace should be lightweight.

- **Accuracy:** A MapReduce job is completed by a number of map and reduce tasks, and these tasks may execute on different machines. Due to the various performance of different machines in the cluster, the execution statistics of each task on different machine have some variances. In order to accurately capture the execution statistics of the MapReduce job, the profiler LTrace should extract the statistics of the tasks executed on the different machines accurately.

Design and Implementation: To realize the above design goals, we design the profiler LTrace as follows.

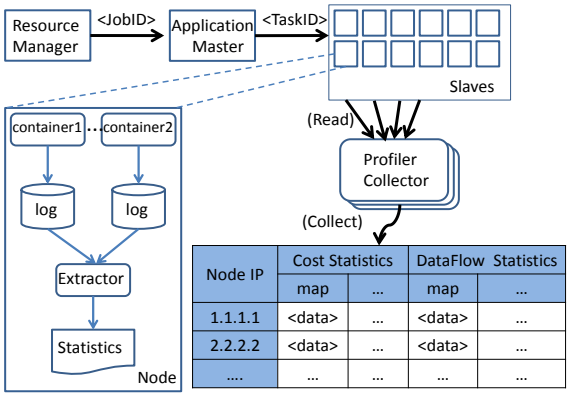

Figure 2. The Data Flow of LTrace

The data flow diagram of the profiler is shown as the figure 2. In order to be a lightweight tools, the LTrace is divided into two separate parts, and these two parts are respectively designed to generate the raw data of data flow and cost when the map and reduce tasks are executing and to off-line extract the execution statistics of the MapReduce job from the raw data. The raw data of the data flow and cost is generated by means of the log system, a small number of logging statements are inserted to record the row data and printed to the local disk when the task is running. Then the log data is extracted to obtain the execution statistics when the execution of the job is complete.

There are multiple MapReduce jobs executed simultaneously on the hadoop cluster and even multiple map and reduce tasks executed simultaneously on the same machine. A MapReduce job consists of multiple map and reduce tasks, and a map or reduce task generates a log file. In order to divide these log files into different partitions that the log files of the same partition are generated by the same MapReduce job, it is essential to add a unique job id to each log. As shown the figure 2, a new MapReduce job is submitted to hadoop cluster, the resource manager allocates the resources for the new jobs and then start the application master which schedules the execution of the map and reduce tasks for the current MapReduce job. The application master receives a unique job id from the resource manager when it starts, and a unique task id is assigned to the start-up map or reduce task. Therefore, each map or reduce task owns a unique task id, and the task id can also identify the all the tasks scheduled by the same MapReduce job. In summary, this task id if should be automatically added to the logs generated by the task.

Apache Log4j, a popular logging system, is applied in the Apache Hadoop. The Log4j provides a convenient feature: the Mapped Diagnostic Context(MDC), and this context can be used to store values that can be displayed in every log. When the process which executes the map or reduce task is started, the task id should be injected to the MDC, and all the threads and child threads can get the task id from the MDC. In order to automatically add the task id to every log, instead of manually adding the task id to each logging statement, a new layout is implemented to redefine the format of the output logs as follows: $< date, thread, taskId, message >$.

As shown in the figure 2, the Extractor, a component of the LTrace, is deployed on all the machines in the cluster, and it off-line extract the execution statistics from the raw log files. The Profiler Collector, another component of the LTrace, is deployed on the one of the machines in the cluster. This collector collects all the execution statistics extracted by the Extractor from the other machines and stores the statistics into the disk files as shown in figure 2.

IV. An Accurate Performance Model

This section introduces a new performance model constructed to accurately predict the execution time of the MapReduce job in the Hadoop cluster.

To construct an accurate performance model, there are some effective methods and principles as follows:

- To accurately predict the execution time of the MapReduce job, this performance model should be able to forecast each execution phase's cost of the job accurately.
- The output of each execution phase is the input of the next execution phase, hence this model also needs to forecast the volume of each phase's output beside the execution cost.
- The execution time of the map and reduce task is not the sum of each phase's cost, due to the presence of parallel that multiple phases execute simultaneously. In order to construct an accurate model, it's necessary to confirm whether the parallel exists between multiple phases, and to predict the overlap time between these phases executed in parallel.

The execution of a MapReduce job consists of the map and reduce tasks, hence the construction of the performance model is divide two sections as follows.

A. The Cost Model For Map Task

The execution of the map task, which is described in the section 2, consists of the following phases: read, map, spill

and merge. The key of our work is to predict the actual cost of map and spill.

Map and Spill. The user-defined map function is executed by the main thread to process the input data, and the spill of the buffer data is performed by the spill thread to sort, combine and write the buffer data sequentially.

(i) Execution Procedure

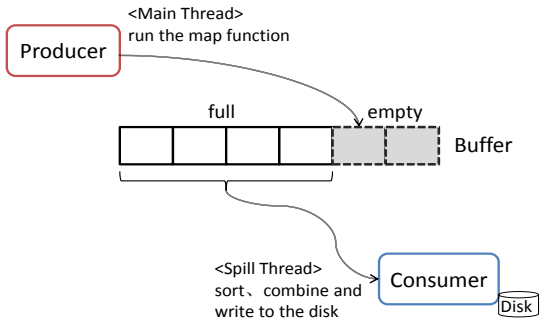

Figure 3. The Interaction between Map and Spill

As shown in figure 3, when the main thread executes the map function, the output of the map function is written into the memory buffer. The spill thread starts to spill the outputs cached in the buffer to the disk file when the used space of the buffer reaches the threshold determined by the related configuration parameter. The main thread continues to execute the map function, when the spill thread spills the data into the disk file. The main thread starts to sleep when the buffer is full and stop sleeping after the data in the buffer is written to disk file.

(ii) The Analysis of Key Effect Factors

- **The cost of map function's execution every time** m_{cs}, and **the number of map function's execution** mn_d. The factor m_{cs} is determined by not only the complexity of the user-defined map function but also the execution environment which consists of the resource consumed by map task and the number of executing threads in the map task.
- **The size of memory buffer** mb_p, and **the threshold at which the spill is triggered** st_p. These two factors determine when to spill.

The total execution time of map and spill is not their linear additivity, due to the parallel execution between map and spill. To predict the total cost of these two phases, we need to estimate not only the individual cost of map and spill, but also their overlap time during which these two phases execute simultaneously. The cost of map is determined by the m_{cs} and mn_d, and the cost of spill is determined by the cost of one spill os_c and the number of spill affected by the mb_p and st_p. If the mb_p and st_p increase, the os_c increases but the number of spill decreases, therefore it's uncertain whether the cost of spill increases or not. Their overlap time is determined by the os_c and the cost of map when spilling mws_c which is also affected by the mb_p and st_p. when the mb_p is fixed, if the st_p increases, the os_c increases but the mws_c might decreases or increases(it's determined by whether the main thread is sleep or not), therefore it's also uncertain whether the overlap time increases or not. In summary, it's challenging to estimate the total execution time of map and spill which is nonlinearly affected by these key factors. We construct the following model to predict the total execution time of map and spill.

(iii) The Construction of Model

We propose a model to predict the total execution time of map and spill as follows. We need to predict the dataflow of these two phases before estimating the cost.

DATAFlOW: The dataflow of these two phases consists of the input and output of each phase. We estimate the output of each phase through the output of last phase and the dataflow statistics captured by the LTrace, for example, the output of map is estimated as follows.

$$mor_d = mn_d \times \frac{\sum_{i=1}^{n} mrs_{ds}^i}{n}$$

where the mor_d is the total number of output records generated by the map function, the mn_d is the number of input records to be processed by the map function, and the mrs_{ds}^i is the records selectivity of the map function performed on the ith machine.

$$mob_d = mib_d \times \frac{\sum_{i=1}^{n} mbs_{ds}^i}{n}$$

where the mob_d is the total size of output bytes generated by the map function, the mib_d is the size of input bytes to be processed by the map function, and the mbs_{ds}^i is the sizes selectivity of the map function performed on the ith machine.

$$morw_d = \frac{mob_d}{mor_d}$$

where the $morw_d$ is the average size of each output record generated by the map function.

Similarly, we estimate the input and output of the spill which consist of the number of records per spill rps_d, the number of output records per spill sor_d, the size of output bytes per spill sob_d, and the number of spills ns_d.

COST: To predict the total execution time of these two phases, we have to estimate the costs of these two phases and their overlap time as follows.

$$mas_c = s_c + ls_c + m_c - otms_c \quad (1)$$

where the mas_c is the total execution time of these two phases, the s_c is the cost of all the spills except the last, the ls_c is the cost of last spill executed by the main thread, the m_c is the cost of the map, and the $otms_c$ is the overlap time between these two phases.

To predict the total execution time mas_c, it's essential to accurately predict the s_c, ls_c, m_c and the $otms_c$.

The $cSpillCost$ is estimated as follows.

$$s_c = ns_d \times os_c \quad (2)$$

where the ns_d is the number of spills executed by the spill thread, and the execution cost per spill os_c which is estimated as follows.

$$os_c = rps_d \times \frac{\sum_{i=1}^{n} s_{cs}^i}{n} + rps_d \times \frac{\sum_{i=1}^{n} c_{cs}^i}{n} + sob_d \times \frac{\sum_{i=1}^{n} w_{cs}^i}{n}$$

where the rps_d, rps_d and sob_d are estimated by the means introduced above, and the s_{cs}^i, c_{cs}^i and the w_{cs}^i are respectively the cost statistics about sorting, combining and writing

The cost of last spill ls_c is estimated as follows.

$$\begin{aligned} ls_c = (mor_d - ns_d \times rps_d) \times \frac{\sum_{i=1}^{n} s_{cs}^i}{n} + (mor_d \\ -ns_d \times rps_d) \times \frac{\sum_{i=1}^{n} c_{cs}^i}{n} + morw_d \times (mor_d \\ -ns_d \times rps_d) \times \frac{\sum_{i=1}^{n} cb_{ds}^i}{n} \times \frac{\sum_{i=1}^{n} w_{cs}^i}{n} \end{aligned} \quad (3)$$

where the ls_c is composed of the costs of sorting, combining and writing the buffer data into a new disk file, and the cb_{ds}^i is the data statistics about combining.

Through the above analysis of the key effect factors, the cost of map function's execution every time is effected by the number of executing threads in the map task. The main thread may execute the map function when the spill thread is spilling the data cached in the buffer into a new file, and the execution speed of the map function is slower due to the simultaneous execution of the spill thread. Therefore, the execution cost of map is divided into two parts as follows.

$$m_c = mws_c + mns_c \quad (4)$$

where the m_c is the total execution time of map, the mws is the execution time of map when the spill thread is running and the mns_c is the execution time of map when the spill thread is stopping.

The spill thread spills the data cached in the buffer in parallel with the main thread which executes the map function, due to the remaining space of the buffer. The main thread starts to sleep when the buffer is full, and the main thread restarts to execute the map function when the spill is completed. Therefore, the estimation of the mws_c and mns_c is complicated, which is closely associated with the remaining space of the buffer.

We assume that there are not enough remaining space in the buffer to hold the output of the map function when the spill thread is executing, and the execution time of the main thread when the spill thread is working is less than the cost of one spill due to the sleep of the main thread. This assumption is judged as follows.

$$isMainThrSleep = mwos_c < os_c$$

where the $mwos_c$ is the cost of the map when the spill thread is executing, and it is estimated as follows,

$$mwos_c = \frac{mb_p \times (1 - st_p)}{(morw_d + 16)} \times \frac{\sum_{i=1}^{n} mws_{cs}^i}{\sum_{i=1}^{n} mrs_{ds}^i}$$

where the mws_{cs}^i is the cost of map function's execution every time when the spill thread is running on the ith machine, and it is captured by the LTrace.

When the above assumption is tenable, the main thread starts to sleep when the buffer is full, and the mws_c and mns_c are estimated as follows.

$$\begin{aligned} mws_c = mwos_c \times (ns_d - 1) + Min(\frac{mb_p \times (1 - st_p)}{(morw_d + 16)}, mor_d \\ -rps_d \times ns_d) \times \frac{\sum_{i=1}^{n} mws_{cs}^i}{\sum_{i=1}^{n} mrs_{ds}^i} \end{aligned}$$

$$\begin{aligned} mns_c = (mor_d - (ns_d - 1) \times \frac{mb_p \times (1 - st_p)}{(morw_d + 16)} - Min(mor_d \\ -rps_d \times ns_d, \frac{mb_p}{(morw_d + 16)} \times (1 - st_p))) \times \frac{\sum_{i=1}^{n} m_{cs}^i}{\sum_{i=1}^{n} mrs_{ds}^i} \end{aligned}$$

where the m_{cs}^i is the cost of map function's execution every time when the spill thread is not running on the ith machine, and it is captured by the LTrace.

When the above assumption is not tenable, the main thread does not sleep due to the enough space of the buffer. The mws_c and mns_c are estimated as follows.

$$mws_c = nmws_d \times \frac{\sum_{i=1}^{n} mws_{cs}^i}{n}$$

$$mns_c = nmns_d \times \frac{\sum_{i=1}^{n} m_{cs}^i}{n}$$

where the $nmws_d$ is the number of map function's execution when the spill thread is running, and the $nmns_d$ is the number of map function's execution when the spill thread is sleeping, and they are estimated as follows.

$$\begin{aligned} nmws_d = \frac{os_c \times n}{\sum_{i=1}^{n} mws_{cs}^i} \times (ns_d - 1) + Min(\frac{os_c \times n}{\sum_{i=1}^{n} mws_{cs}^i}, \\ \frac{(mor_d - ns_d \times rps_d) \times n}{\sum_{i=1}^{n} mrs_{ds}^i}) \end{aligned}$$

$$nmns_d = mn_d - nmws_d$$

At last, the $otms_c$ is estimated as follows.

$$otms_c = mws_c \quad (5)$$

B. The Cost Model For Reduce Task

The execution of the map task, which is described in the section 2, consists of the following phases: shuffle, merge, reduce and write. The key of our work is to predict the effective cost of shuffle.

Shuffle. The reduce task reads the corresponding outputs of all the map tasks from the remote machines in the cluster.

(i) Execution Procedure

As shown in figure 4, multiple copy threads of the reduce task are started to simultaneously fetch the corresponding outputs of map tasks from the various machines When there are multiple outputs generated by different map tasks on a

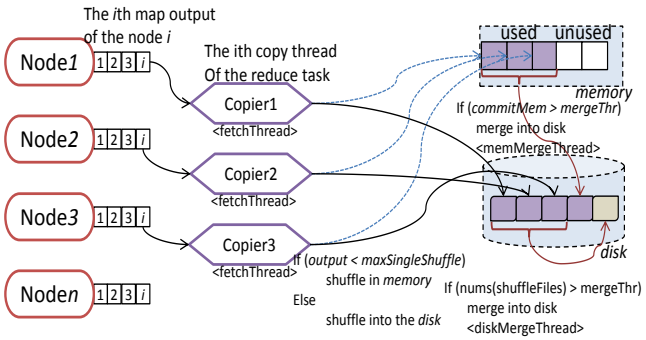

Figure 4. The Execution Procedure of Shuffle

machine, only one of these copy threads sequentially fetches these outputs from the target machine.

When the copy thread starts to fetch the output of the completed map task, the output can be written to the memory buffer or a new disk file, which is determined by the way as shown in figure 4. A merge thread is started, when the used space of the memory buffer or the number of disk files reach the threshold.

(ii) The Analysis of Key Effect Factors

- **The number of the effective copy threads** nc_p, the nc_p is the actual number of copy threads which simultaneously fetch the outputs of map tasks from nc_p machines.
- **The output of the map task** mo_d, and **the max single shuffle limit** $mssl_p$. These two factors determine whether the map task's output is fetched into the buffer or disk.
- **The threshold at which the memory merge is triggered** mt_p, and **the number of disk files at which the disk merge is triggered** sf_p. These two factors affect when the merge thread merge the buffer data or the disk files into a new file.

(iii) The Construction of Model

We propose the simulation approach to predict the whole shuffle time of the reduce task.

Algorithm 1 The simulation of shuffling into the buffer

Require: $mapOutputs$, $numCopies$
Ensure: $cShuffleTime$

function CSHMCOST($mapOutputs, numCopies$)
  $T_c \leftarrow 0, T_m \leftarrow 0, T_d \leftarrow 0$
  $haveShu \leftarrow 0, isM \leftarrow 0, isF \leftarrow 0$
  $Outs_m \leftarrow null, Outs_d \leftarrow null$
  $numOuts_{mtod} \leftarrow 0, copyDests \leftarrow null$
  $indexDest \leftarrow 0$
  for $i = 0 \rightarrow numCopies - 1$ **do**
    if $i < length[mapOutputs]$ **then**
      $copyDests[i] \leftarrow mapOutputs[i]$
      $indexDest \leftarrow indexDest + 1$
    else
      $copyDests[i] \leftarrow null$
    end if
  end for
  while !ISNULL($mapOutputs$) **do**
    for $t = 0 \rightarrow numCopies - 1$ **do**
      $out \leftarrow copyDest[t]$
      if $out \neq null$ **and** $T_c >= T_s[out]$ **then**
        $nums[out] \leftarrow nums[out] - 1$
        $Outs_m \leftarrow Outs_m + out$
        $haveShu \leftarrow 1$
        if ISMERGE($MapOuts_m$) **then**
          $isM \leftarrow 1$
          $numOuts_{mtod} \leftarrow size[Outs_m]$
        end if
        if $nums[out] == 0$ **then**
          UPDATECOPYDEST($indexDest$, $mapOutputs$)
        end if
        if ISFULL($MapOuts_m$) **then**
          $isF \leftarrow 1$
          exit
        end if
      end if
    end for
    if $havaShu == 1$ **then**
      $T_c \leftarrow T_c +$ SHUCOST($mapOutputs[0]$)
      $haveShu \leftarrow 0$
      if $isM == 1$ **then**
        $T_m \leftarrow$ MAX(T_c, T_m)
        $T_m \leftarrow T_m +$ MEMMERGE($MapOuts_m, numOuts_{mtod}, Outs_d$)
        if ISDISKMERGE($Outs_d$) **then**
          $T_d \leftarrow$ MAX(T_d, T_m)
          $T_d \leftarrow$ DISKMERGE($Outs_d$) $+ T_d$
        end if
        $isM \leftarrow 0$
      end if
      if $isF == 1$ **then**
        REMOVEMEMOUTS($Outs_m$, $numOuts_{mtod}$)
        $T_c =$ MAX(T_c, T_m)
        $isF \leftarrow 0$
      end if
    else
      $T_{new} \leftarrow$ LASTEDCLOCK($mapOutputs$)
      $T_c \leftarrow$ MAX(T_c, T_{new})
    end if
  end while
  $cShuffleTime \leftarrow$ MAX(T_c, T_m, T_d)
end function

In the algorithm 1, the $mapOutputs$ is the set of the map task's outputs on all the machines, each element of this set records the number and the finish time of map tasks simultaneously executing on the one machine. The T_c, T_m and T_d respectively are the clock of copy thread, memory merge thread and the disk merge thread, and these clocks record the start or end time of the related trigger event, for example, the T_m records the start and end time of the memory merge executed by the memory merge thread. When a event is trigger by one thread, the start time of the event is updated to be the current clock of that certain thread, and the end time of the event is the sum of this event's cost and the start time. For example, if the buffer data size reach the memory merge threshold after one copy thread fetches the data into the buffer, the memory merge event is trigger, the start time of this memory merge T_m is updated to be the copy thread's clock T_c, and then the T_m increases by the cost of this merge thread, which presents the end time of this event. The value of isM indicates whether it's time to start to memory merge thread to merge the buffer data into a new file, and the isF indicates whether the buffer is full. The $Outs_m$ and $Outs_d$ are the outputs of map tasks stored in the memory and disk respectively. The value of $haveShu$ indicates that whether there are map tasks' outputs to be fetched at the moment, if the $haveShu$ is 0, then the T_c needs to be updated as the latest finish time of all the remaining map tasks. The $cShuffleTime$, the cost of shuffle, is the maximum of T_c, T_m, T_d. It is worth to note that our algorithm can predict the effective shuffle time of the reduce task, when there are multiple waves of map tasks and reduce tasks simultaneously running on this cluster.

Similar to the algorithm 1, we can use the similar method to calculate the cost of the shuffle, when the map tasks' outputs are written into the disk. The T_c and T_d respectively are the clock of copy thread and disk merge thread, and the maximum of the T_c and T_d is the shuffle cost $cShuffleTime$.

V. Experiment

In this section, we conduct the following experiments to evaluate the prediction accuracy of the Mr. Prophet, and we compare the effectiveness of this new model with the What-IF [3] under a variety of input data and jobs.

A. Experiment Setup

We deploy a Hadoop cluster as the experimental environment to evaluate the Mr. Prophet. The Hadoop cluster with 16 machines is deployed, one of these machines is deployed as the master which runs the Resource Manager and the NameNode, and the other machines are deployed as the slave which runs the Node Manager and the Data Node.

Each machine in the cluster is with 16×2.40GHz Processors and 50GB memory. The Hadoop version is 2.6.0, and the YARN(Yet Another Resource Negotiator) [9] manages(allocate and deallocate) the resource of the Hadoop cluster. There are totally 240 cpu cores and 480 GB memory in the cluster, each machine in the cluster provides 16 cpu cores and 32 GB memory. Therefore, 240 map or reduce tasks can execute simultaneously, due to a map or reduce task requires 1 cpu core and 2 GB memory by default.

We use the HiBench [10] as the benchmark suit, this benchmark suit contains 13 workloads, and we choose three typical workloads as our experimental workloads which are as follows. Wordcount(which counts the occurrence of each word in the input data.), Terasort(which sorts the input data) and the Pagerank(which is MapReduce implementation of Pagerank algorithm.).

B. The Accuracy of the New Model

To validate the prediction accuracy of the Mr. Prophet, the MapReduce jobs(Wordcount, Terasort and Pagerank) are required to execute on the Hadoop cluster once, and meanwhile the LTrace extracts the statistics of the MapReduce jobs, which is the prerequisite for the prediction. Then the execution time of these jobs with various input data is predicted by the new model and What-IF respectively as follows.

To illustrate the effectiveness of the new model, we establish the following evaluation indicator.

$$I_{accu} = \frac{|T_{wf} - T_a|}{T_a} - \frac{|T_{new} - T_a|}{T_a} \tag{6}$$

where the I_{acct} is the indicator value which represents the improvement of the Mr. Prophet's prediction accuracy relative to the What-IF, the T_a represents the actual execution time of the MapReduce job, the T_{wf} is the execution time predicted by the What-IF, and the T_{new} is the execution time predicted by the Mr. Prophet.

As shown in the equation (6), the $\frac{|T_{wf}-T_a|}{T_a}$ is the relative error of the What-IF's prediction accuracy, and the $\frac{|T_{new}-T_a|}{T_a}$ is the relative error of the new model's prediction accuracy. The decrease of relative error illustrate that the prediction accuracy of the new model is higher than the What-If's, and the higher the I_{accu}, the better the effectiveness of the Mr. Prophet relative to the What-If.

Table 3 orderly shows the job's actual execution time, the job's execution time respectively predicted by the Mr. Prophet and What-IF, and the I_{accu} which illustrates the enhancement of our Mr. Prophet relative to the What-IF under the various size of input data. As shown in the table 3, all the I_{accu} of various jobs under the different input data are greater than 0, which illustrates that the prediction accuracy of the Mr. Prophet is better than the What-IF to some extent. For the application Wordcount, when the size of the job's input data reaches the 10 GB, all the I_{accu} are greater than 10%, which shows that our Mr. Prophet owns much stronger forecasting ability than the What-IF

Table I
THE PREDICTION ACCURACY OF THE MAPREDUCE JOB'S EXECUTION TIME

Job Name	Input (GB)	Actual (sec)	Mr. Prophet (sec)	What-IF (sec)	I_{accu} (%)
wordcount	2	70	67	75	3.0
wordcount	10	123	118	142	11.9
wordcount	30	301	292	339	10.0
wordcount	50	469	491	541	10.6
wordcount	80	742	772	850	10.5
wordcount	100	920	960	1082	13.2
terasort	30	119	112	132	5.0
terasort	50	142	130	162	5.6
terasort	100	305	292	336	5.9
terasort	200	1020	960	1150	6.8
pagerank	31	279	266	313	7.5
pagerank	51	350	337	385	6.5
pagerank	92	645	624	728	9.6

especially for the large input data. This is because when the size of input data is larger, there are more data the reduce task have to process, and then there are more data to shuffle, which is simultaneously processed by the multiple copy threads and merge threads, therefore there are much more overlap time between the copy and merge phases, which can be captured by the Mr. Prophet. For the applications Terasort and Pagerank, all the I_{accu} are less than 10%, it is because these jobs have over 100 reduce tasks, and the data processed by each reduce task is relatively small amount. Despite all this, the prediction accuracy of our Mr. Prophet is above 94% for the Terasort and Pagerank. It is noteworthy that all the execution time predicted by the What-IF are greater than the actual execution time, which demonstrates that the executing procedure of Hadoop MapReduce job is extremely complex and it is not simply executed step by step. Therefore, it is necessary to find out the parallels between multiple different phases and quantify the overlap time of these parallel phases.

The execution of MapReduce job consists of the map tasks and reduce tasks, the prediction accuracy of the map and reduce tasks' execution time is shown as the follows.

Table II
THE PREDICTION ACCURACY OF THE MAP TASK'S EXECUTION TIME

Job Name	Input (GB)	Actual (sec)	Mr. Prophet (sec)	What-IF (sec)	I_{accu} (%)
wordcount	2	53.8	54.7	61.8	13.2
wordcount	10	53.1	54.7	61.8	13.4
wordcount	30	52.3	54.7	61.8	13.6
wordcount	50	55.5	54.7	61.8	12.7
wordcount	80	54.4	54.7	61.8	13.1
wordcount	100	53.9	54.7	61.8	13.2
terasort	30	32.6	28.4	29.2	N
terasort	50	33.5	28.4	29.2	N
terasort	100	31.2	28.4	29.2	N
terasort	200	35.9	28.4	29.2	N
pagerank	31	81.8	76.2	84.2	N
pagerank	51	80.2	76.2	84.2	N
pagerank	92	82.4	76.2	84.2	N

Table 4 shows the map task's actual execution time, the map task's execution time respectively predicted by the Mr. Prophet and What-IF, and the I_{accu}. The input data of the map task has nothing to do with the input data of the MapReduce job, one map task processes a block of the HDFS, and the size of each block is uniform 128 MB by default, which can be specified by the related configuration parameter. Therefore, for the each application, the execution time of all the map tasks predicted by the model is the same due to the same block size of the HDFS. As shown in the table 4, for the application Wordcount, all the I_{accu} are greater than 10%, and the prediction accuracy of our Mr. Prophet for the map task is above 96%. However, for the Terasort and Pagerank, our Mr. Prophet can't predict accurately the execution time of the map task, this is because there are extra disk I/O latency during the phase of merge, the map task needs to create a new file and merge the relevant data of all the spill files into this new file for each reduce task, there are over 200 reduce tasks for these two jobs and 16 map tasks running simultaneously on one machine, therefore the disk I/O is so busy ,due to the frequent creation and access to various small files, which increases the latency of the disk I/O request response. Unfortunately, our Mr. Prophet can't capture this latency of disk I/O request response, and the What-IF also doesn't not have this ability. The executing speed of the map task is affected by the number of reduce task to some extent, and it is a valuable experience to tuning the performance of the MapReduce jobs.

Table III
THE PREDICTION ACCURACY OF THE REDUCE TASK'S EXECUTION TIME

Job Name	Input (GB)	Actual (sec)	Mr. Prophet (sec)	What-IF (sec)	I_{accu} (%)
wordcount	2	13.9	12.6	12.9	N
wordcount	10	66.5	63.5	76.6	10.7
wordcount	30	245.1	239.8	277.4	11.1
wordcount	50	408.4	436.7	509.7	17.6
wordcount	80	680.3	717.4	835.2	17.3
wordcount	100	863.0	905.9	1072.2	19.7
terasort	30	52.4	55.4	62.5	13.5
terasort	50	69.7	72.8	86.1	19.1
terasort	100	165.7	172.4	202.1	17.9
terasort	200	260.5	276.3	329.2	20.3
pagerank	31	108.2	112.1	128.2	14.8
pagerank	51	175.1	184.9	208.3	13.4
pagerank	92	302.7	319.5	371.3	17.1

Table 5 shows the reduce task's actual execution time, the reduce task's execution time respectively predicted by the Mr. Prophet and the What-IF, and the I_{accu}. The input data of the reduce task has a positive correlation with the MapReduce job's input data when the number of reduce tasks is invariant, there are more map tasks when there are more input data processed by the MapReduce job, therefore there are more output data of all the map tasks to be processed by each reduce task. As shown in the table 5, for

the application Wordcount, our Mr. Prophet doesn't show the stronger ability than the What-IF, when the size of input data is 2 GB, this is because that all the 16 map tasks simultaneously execute on one machine, and the map tasks' outputs to be process by the reduce task are small, therefore there is only one copy thread fetching the map tasks' outputs and not merge thread merging the data in the buffer into the disk. For these three application, when the size of input data is larger, the I_{accu} is higher, which indicates our Mr. Prophet has stronger predictive ability than the What-If, especially when the input data is large. This is because there are more map tasks' outputs to be simultaneously copied into the reduce task by multiple copy threads, and meanwhile the merge threads are started to merge the buffer data or the multiple disk files into the disk file. The overlap time between the copy and merge phases increases rapidly, when the input data is getting larger, therefore it is so necessary to quantify the overlap time, and the unfortunate reality is that the What-IF doesn't have the ability to capture these overlap time. These experiments shows that our Mr. Prophet can make up for the defect of the What-IF.

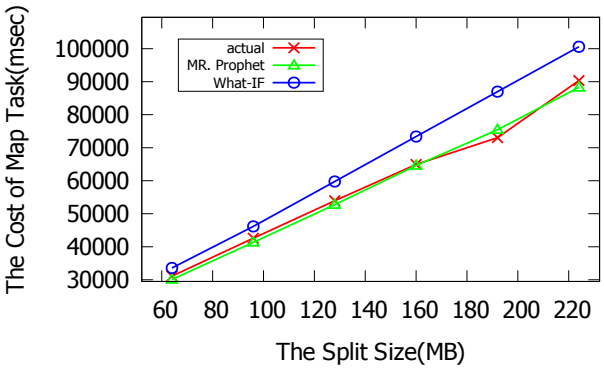

Figure 5. The prediction for the map tasks with various split size

To more fully illustrate the effectiveness of the Mr. Prophet, we do an extra experiment to compare the prediction accuracy of the Mr. Prophet and What-IF for the map tasks with various input data. We adjust the map task's input size by changing the configuration parameter dfs.block.size, and the parameter value is set from 64 to 224 MB. As shown in the figure 5, our Mr. Prophet owns the strong ability to accurately predict the executing time of the map tasks with various input size. As expected, we can find that all the map tasks' execution time predicted by the What-IF are greater than the actual execution time, and the gap is more and more obvious when the input size increases. This is because the spill thread is started to spill the buffer data into the disk file when the size of buffer data reaches the spill threshold, and the main thread still executes the map function due to the remain free space of the buffer, hence there is overlap time between the map and spill phases. Unfortunately, the What-IF can not capture this overlap time between the map and spill phases. When the input size increases, there are more outputs of the map function to be spilled from the buffer into the disk file, which increases the overlap time accordingly. So what, our Mr. Prophet holds the strong ability to capture these overlap time, which improves the predictive accuracy greatly.

VI. Related Work

What-IF [3] is a performance prediction engine, which is applied to the self-tuning system for the MapReduce called Starfish[**?**]. The authors propose a model to predict respectively the execution time of the map and reduce task, but it doesn't consider the parallelism between multiple phases inside the map or reduce task. Shi [4] proposes a novel Producer-Transporter-Consumer(PTC) model, which is adopted by the MRTuner to optimize the configuration parameters of MapReduce jobs. The model address the issues of inter-task parallelization, and this issue occurs when there are reduce and map tasks simultaneously executing. At this moment, the shuffle phase of the reduce task is in parallel with the map task, our Mr. Prophet also holds the ability to predict the effective time of this reduce task's shuffle phase, but the PTC can't capture the multiple phases' overlap time.

There are other researches to predict the MapReduce job's execution time. MRPerf [11] is the existing simulators which predict the execution time of MapReduce job through the simulation. MRPerf provides the fine-grained simulation of the MapReduce job at sub-phase level like our Mr. Prophet. However, the MRPerf has to take a relatively long time to predict the MapReduce job's cost due to the use of external network simulator to predict the cost of data transfers. MRShare [12] is a framework to transform a batch of queries into a new batch that will be executed more efficiently, the authors proposes a cost model for this optimization, but this cost model is so simplified and it just considers the total cost of read, sort and write. The related researches[13][14][15] propose some other approaches to predict the cost of MapReduce jobs, but they don't find the issue of phases parallelism.

VII. Conclusion

This work presents a performance model of MapReduce Applications with Apache Hadoop, which is useful for making an optimal configuration of Hadoop for improving the application performance. Different from existing work, we take into account the parallelism between the adjacent processing phases. With a clear understanding of the MapReduce implementation by reading source code of Hadoop, we give an anatomy of MapReduce job processing. It shows which specific phases a MapReduce job must go through and which adjacent phases are run in parallel for performance consideration. Then we come up with the performance models for Map and Reduce respectively, where

we particularly address the issue of predicting the execution time when two execution phases overlap with each other. As some parameters in the performance models rely on the characteristics of applications themselves and the specific underlying resources, we design a light-weight instrumentation tool, LTrace, to capture the application and resource specific parameters by running an application with LTrace for only one time. We have conducted a set of experiments with HiBench workload including WordCount, TeraSort and PageRank jobs. Experimental results with different size of data demonstrate that our model can achieve a prediction accuracy as high as over 94%, which greatly outperforms the comparing approach.

In future, we will continue to work on how to take advantage of our performance models to generate an optimal configuration for MapReduce applications. Beyond that, though our model works well for the batch jobs, we believe it is worthwhile to consider the commonly used query jobs. The processing of query jobs depends on the specific query constraints provided by users, which usually involves complex data operations and many map/reduce tasks. This can pose new challenges for the generality of the performance model. Our future work will concern this issue by incorporating other benchmarks such as TPC-DS[16] and BigBench [17].

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters." *In Proceedings of Operating Systems Design and Implementation (OSDI*, vol. 51, no. 1, pp. 107–113, 2004.

[2] T. White, "Hadoop : the definitive guide," *O?reilly Media Inc Gravenstein Highway North*, vol. 215, no. 11, pp. 1 – 4, 2010.

[3] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *Proc of the Vldb Endowment*, vol. 4, pp. 1111–1122, 2011.

[4] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang, "Mrtuner: a toolkit to enable holistic optimization for mapreduce jobs," *Proceedings of the Vldb Endowment*, vol. 7, no. 13, pp. 1319–1330, 2014.

[5] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, "Mronline: Mapreduce online performance tuning," in *International Symposium on High-Performance Parallel and Distributed Computing*, 2014, pp. 165–176.

[6] H. Herodotou, "Hadoop performance models," *Computer Science*, 2011.

[7] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, pp. 1–14.

[8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[9] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, and S. Seth, "Apache hadoop yarn: yet another resource negotiator," in *Symposium on Cloud Computing*, 2013, pp. 1–16.

[10] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *IEEE International Conference on Data Engineering Workshops*, 2010, pp. 41 – 51.

[11] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in mapreduce setups," *Modeling Analysis Simulation of Computer Telecommunication Systems Mascots*, pp. 1–11, 2009.

[12] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "Mrshare: Sharing across multiple queries in mapreduce." *Proceedings of the Vldb Endowment*, vol. 3, no. 1, pp. 494–505, 2010.

[13] G. Song, Z. Meng, F. Huet, and F. Magoules, "A hadoop mapreduce performance prediction method," in *HPCC*, 2013, pp. 820 – 825.

[14] X. Lin, Z. Meng, C. Xu, and M. Wang, "A practical performance model for hadoop mapreduce," in *IEEE International Conference on CLUSTER Computing Workshops*, 2012, pp. 231–239.

[15] F. Teng, L. Yu, Magoul, and S. Frederic, "Simmapreduce: A simulator for modeling mapreduce framework," in *Ftra International Conference on Multimedia and Ubiquitous Engineering, Mue 2011, Crete, Greece, 28-30 June*, 2011, pp. 277–282.

[16] R. O. Nambiar and M. Poess, "The making of tpc-ds," in *International Conference on Very Large Data Bases, Seoul, Korea, September*, 2006, pp. 1049–1058.

[17] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H. A. Jacobsen, "Bigbench: towards an industry standard benchmark for big data analytics," in *ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1197–1208.