A Network Load Perception Based Task Scheduler for Parallel Distributed Data Processing Systems

Zhuo Tang, *Member, IEEE*, Zhanfei Xiao, Li Yang, Kailin He, and Kenli Li

Abstract—In parallel distributed data processing frameworks like Spark and Flink, task scheduling has a great impact on cluster performance. Though task Scheduling has proven to be an NP-complete problem, a large number of researchers have proposed many heuristic rules to obtain approximate optimal solutions. But most of them ignore the fact that the resource requirements of tasks are dynamically changing during its runtime. Considering the overall task entire lives, the CPU utilization is often lower during the data transfer. Especially for most distributed data processing platforms, data transmission is time-consuming, which usually resulting in low overall CPU utilization. Similarly, network throughput during task calculations is also low in some cases. In this article, we propose a network load variation perception based heuristic task scheduling algorithm, and based on this implement a dual-phase pipeline task scheduler (D2PTS) from the perspective of dynamic resource requirements that aims at maximizing cluster resource utilization, as a supplement to existing data-parallel frameworks. D2PTS divides the states of task into two phases: network-intensive and network-free. To improve the overall resource utilities, this article proposes different algorithms to evaluate the execution time of network sensitive and network free phases respectively. When an executing task is in the network-free phase, D2PTS can additionally schedule a new network-intensive task at the right time. Under this scheduling policy, the two tasks sharing the same CPU core can be executed as a coarse-grained pipeline. This execution method can start tasks earlier and improve resource utilization. Finally, we have implemented our model prototype on Spark 2.4.3 and conducted a number of experiments to evaluate the performance of our model. Experimental results show that D2PTS can not only minimize application makespan, but also improve resource utilization.

Index Terms—Dynamic resource requirements, dual-phase pipeline, Spark, task scheduling

1 Introduction

With the rapid development of the Internet and the popularity of data-intensive applications such as search engines and social networks, the scale of data has expanded dramatically. In order to meet the challenges of data processing in the era of big data, a new type of massive data processing platform featuring that moving computation towards data is cheaper than moving data towards computation has emerged, like MapReduce [1]. Hadoop [2], Spark [3] and Flink [4] are products based on the thought of MapReduce. Compared with Hadoop, Flink and Spark, introduce the concepts of DataSet and Resilient Distributed DataSet (RDD) [5] respectively, as memory-based processing frameworks.

In these frameworks, a job can be divided into one or more stages. To facilitate parallel processing, each stage contains multiple tasks to process a subset of the input data in parallel. The problem to be solved by task scheduling is how to allocate n independent tasks to m available resources, and achieve the least time to complete all tasks. But this problem has proven to be an NP-complete problem [6]. A large number of researchers have studied this and propose many heuristic rules to obtain approximate optimal solutions. [7], [8] are devoted to reducing network transmission across racks, [9], [10] based on the assumption that the map task is determined by the position of the input data, considering the placement of the reduce tasks, [11] considers the placement of both map tasks and reduce tasks in cloud environment, [12] optimizes the internal task scheduling mechanism of distributed data processing systems such as Hadoop/Spark. Several scheduling strategies (e.g., FIFO, FAIR) employe techniques like delay scheduling [13] to improve the data locality.

However, most above scheduling mechanisms ignore the fact that the resource requirements of tasks change over time, but statically set the resource requirements of tasks, such as the number of CPU cores, before tasks are scheduled. In fact, the task may be network-intensive for a certain period of time, and CPU-intensive for another during its runtime, resulting in serious waste of resources.

In most existing data-parallel frameworks, such as Spark, task scheduling is triggered by the advent of idle resources. Whenever the executor resource is idle, task scheduler will

- *Zhuo Tang is with Hunan University, Changsha, Hunan 410082, China, and also with National Supercomputing Center in Changsha, Changsha, Hunan 410082, China. E-mail: ztang@hnu.edu.cn.*
- *Zhanfei Xiao, Li Yang, Kailin He, and Kenli Li are with Hunan University, Changsha, Hunan 410082, China. E-mail: {zfeixiao, klinhe, lkl}@hnu.edu.cn, yanglixt@126.com.*

Manuscript received 11 May 2021; revised 1 Dec. 2021; accepted 1 Dec. 2021. Date of publication 6 Dec. 2021; date of current version 7 June 2023.
This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1701400, the National Natural Science Foundation of China under Grants 92055213, 61873090 and L1924056, the Guangdong Province research and development plan project in key area under Grant 2020B0101100001 and the China Knowledge Centre for Engineering Sciences and Technology Project under Grant CKCEST-2021-2-7.
(Corresponding author: Li Yang.)
Recommended for acceptance by J. Zhai.
Digital Object Identifier no. 10.1109/TCC.2021.3132627

2168-7161 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

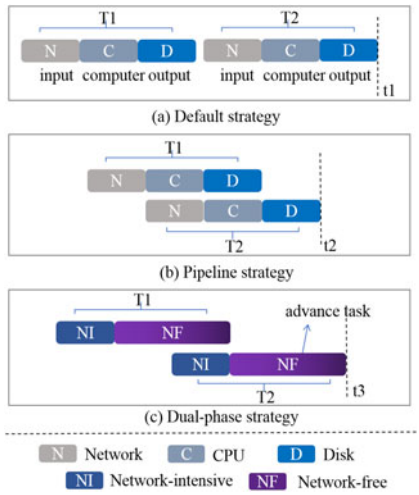


Fig. 1. Task execution model on a single CPU core.

select the appropriate tasks to schedule to that executor to run or wait for the arrival of the more appropriate idle executor. By default, the total number of tasks that can run concurrently in all executors is equal to the number of CPU cores. This means that the maximum value of the total number of tasks running simultaneously is the total number of CPU cores. When the number of partitions is large than the total number of tasks running simultaneously, tasks will be executed in batches. As shown in Fig. 1a, T_2 will not start until T_1 terminates. In the real data-parallel processing environment, we observe that a task requires multiple types of resources such as CPU, network during its execution, and its demand for resources is time-varying. A task typically undergoes several phases, such as data input, computation, and results output, and the main resource requirements of a task are different at different phases. Classically, the main resource requirements of a task are described as follows: network in the input phase, CPU in the compute phase and disk in the output phase. However, the resource requirements of tasks, such as number of CPU cores, are determined statically at the time of task scheduling. The problem is that a CPU core is not always fully used despite a task being assigned to it.

In order to verify that the CPU and network [14] are underused most of the time, we sample CPU and network usage when a Spark $PageRank$ job is running. Figs. 2 and 3 show the sampling results. The average CPU utilization of $PageRank$ is 62%, and the average throughput of $PageRank$ is 12.52MB/s. There are two reasons for this: (1) turnaround time between consecutive tasks. There is a time interval

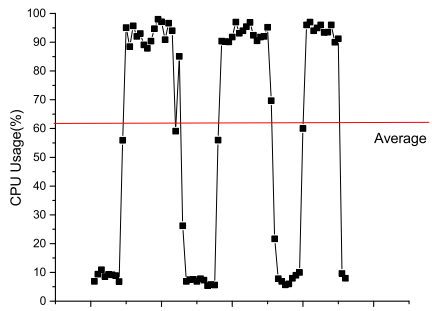


Fig. 2. The utilization of CPU.

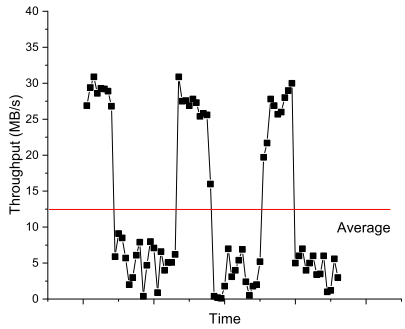


Fig. 3. The utilization of throughput.

between the resource becoming idle and the task being scheduled to the corresponding resource. (2) the task requires only light CPU resources when it is in the input and output phases, and only a small amount of data needs to be transmitted through the network in the calculation phase.

Considering dynamic resource requirements during task runtime, intuitively, we hope that we can simulate the instruction pipeline scheduling to schedule tasks like Fig. 1b for fuller use of resources, making multiple tasks at different phases run on the same CPU core. However, when we directly assign multiple tasks to a CPU core, task pipeline execution might lead to resource competition resulting in performance degradation. In other words, it is inefficient to simply launch multiple tasks to a single CPU core at the same time.

With this intuition, we implement a dual-phase pipeline task scheduler (D2PTS), as an novel scheduling mechanism for the actual distributed systems. D2PTS is suitable for most parallel distributed computing frameworks. In the actual framework, we chose Spark to describe how to integrate D2PTS into a distributed framework. We divide the execution status of the task into two phases: network-free (no network needed) and network-intensive (network needed). When an executing task is in the network-free phase, schedule an additional new network-intensive task at the right time. These two tasks sharing the same CPU core are executed as a coarse-grained pipeline as shown in Fig. 1c. When T_1 is in the network-free phase, additionally schedule T_2 that needs to fetch data. Therefore, when the execution of T_1 terminates, T_2 enters the network-free phase. This execution model not only avoids the competitive use of resources but also reduces the idle period of resources, thereby improving the performance of the entire cluster. The new network-intensive task is equivalent to being scheduled in advance by D2PTS, referred to as an *Advance* task henceforth in this paper. Towards this end, our contributions can summarize as follows:

- We implement task data volume statistics including both global and remote data. Data transmission time and remaining calculation time of tasks are quantized based on the amount of data using different algorithms for more efficient task scheduling.
- By closely considering the data transmission time and remaining calculation time, We design a dual-phase pipeline task scheduler and solve the task scheduling using efficient heuristics, which lead to significant benefits in practice.

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

- We have implemented our model prototype on Spark 2.4.3 and conducted a number of experiments to evaluate the performance of our model. The experiment proves that it can improve resource utilization and shorten job completion time.

The rest of the paper is organized as follows: Section 2 introduces the related work on task scheduling based on MapReduce. Section 3 introduces the overall model design of D2PTS. Section 4 describes the detailed implementation of D2PTS. The experimental results and evaluations are presented in Section 5. Section 6 concludes the paper.

2 Related Works

Task scheduling for MapReduce programs has become a hot research area after the popularity of MapReduce paradigm. From the previous work, the improvement of task scheduling performance mainly comes from the following four aspects: reducing communication during shuffle phase, especially cross-rack network traffic [7], [15], [16], speculative execution of tasks [17], [18], [19], task scheduling in heterogeneous environments [20], [21], [22], improving resource utilization [23], [24], [25]. However, most work does not fully account for the dynamic nature of the task's resource requirements. The direct impact is the inadequate use and imbalance of resources. As stated in Section 1, the maximum value of the total number of tasks running simultaneously is the total number of CPU cores. When the number of partitions is large than the total number of tasks running simultaneously, tasks will be executed in batches. The new task can be scheduled for execution only if the former task finished executing and released the CPU core. The problem is that a CPU core is not always fully used despite a task being assigned to it.

There are also many studies dedicated to solving this problem. Jiang *et al.* [23] design a network-aware task scheduler called *Symbiosis* that predicts resource imbalance before launching tasks, and corrects such imbalance by co-locating computation-intensive (with data locality) and network-intensive (without data locality) tasks in the same executor process. *Symbiosis* does not explicitly modify the core number of the executor, but uses a "forging" mechanism to launch additional tasks. Whenever task scheduler launches a network-intensive task, *Symbiosis* will fill the same executor with a computation-intensive from the pending tasks at the same time. However, it distinguishes a task between computation-intensive and network-intensive through data locality, which easily cause competition for resources. Moreover, the tasks scheduled by *Symbiosis* are limited to map task, resulting in ineffective in most cases.

Li *et al.* [26] propose a scheduling method that considers the dynamic change of task requirements called BETS. BETS introduces the concept of *duty cycle* to represent the actual demand of the task on CPU. Low *duty cycle* means that it is more likely to share the CPU with other tasks. BEST needs real-time monitoring of resources and updates the value of *duty cycle*. However, We must realize that the overhead of real-time monitoring is not negligible.

Bae *et al.* [27] propose a workload-aware task scheduling method which dynamically sets the following two parameters at runtime: number of data partitions specifying task granularity, and numbers of tasks per each executor specifying the degree of parallelism in execution. This is a method of dynamically modifying the number of tasks that can be run simultaneously in an executor instead of statically based on the number of executor CPU cores, but ignores the dynamic change of task requirements.

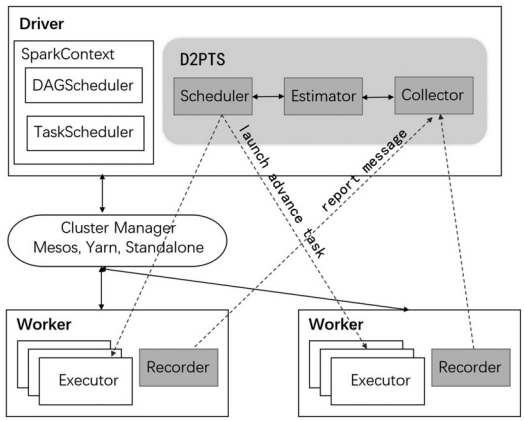


Fig. 4. The architecture of D2PTS.

Xu *et al.* [28] propose a scheduling method, namely *Prophet*, which takes the dynamics on the resource demands into account from executor scheduling perspective. *Prophet* can dynamically avoid severe resource contention, improve cluster resource utilization and minimize application makespan. Similarly, considering requirements dynamics from task scheduling perspective is also very effective.

In brief, considering the runtime resource requirements diversification of tasks can help us schedule tasks more efficiently. The two challenges our model needs to address are as follows: (1) how to quantify the time the task is in the network-intensive and network-free phases, respectively and (2) when should a network-intensive task be scheduled to share the CPU core with a network-free task. In the remainder of this paper, we will give detailed solutions.

3 System Overview

In this section, we present the design of D2PTS that co-locates two tasks on the same slot to achieve pipeline execution and improve resource utilization based on the relationship between data transmission time and remaining calculation time. Data transmission time is a quantitative indicator of the length of time a task will be in the network-intensive phase, while remaining calculation time represents the remaining time of a task in the network-free phase. These two quantified indicators are used to implement our dual-phase pipelined task scheduler. In order to describe our work more vividly, we build our model on Spark 2.4.3. Fig. 4 shows the system architecture of D2PTS. We designed and implemented D2PTS with the benefit of making it easier to use the distributed communication mechanism and task management scheduling framework already in Spark. And this design can be easily applied to other distributed parallel processing systems such as Flink.

When a Spark application is submitted, *Driver* is created and registered with *Master*. *SparkContext* applies for resources from *Cluster Manager* (e.g., Mesos [29], Yarn [30], Standalone) to create *Executors*. Spark models RDD relationships in the form of directed acyclic graph (DAG). *DAGcheduler*

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

will divide the DAG into multiple stages according to the shuffle transformations of RDD, and generates a task set based on a ready stage (its parent stages all finished) and submits the task set to the *TaskScheduler*. *TaskScheduler* is responsible for task scheduling, and finally launches tasks on specific *Executor* for executing. In order to implement our dual-phase pipeline scheduler, based on the original scheduling mechanism, we redeveloped the following four new components and successfully adapted them to the existing framework:

- Scheduler: It is mainly responsible for finding *Advance* tasks and dispatching them to the *Executor*, It communicates with the Estimator before making its scheduling decisions.
- Estimator: It is responsible for obtaining information from the Collector, calculating the data transmission time and the remaining calculation time of the tasks.
- Collector: It is responsible for collecting the messages sent by Recorder and calculating task data volume include both global and remote data.
- Recorder: It is responsible for reporting the status of the tasks. When the task completes the data transmission, it reports the current system time to the Collector in the form of tuple <taskid, time>.

Among them, Scheduler, Estimator, Collector are embedded in *Driver*. The Recorder runs on the *Worker* node running executors. We leave the detailed implementation details to Section 4.

4 Dual-Phase Pipeline Task Scheduling

In this section, we introduce a dual-phase pipeline task scheduler. The purpose of D2PTS is to achieve the pipeline execution of network-intensive task and network-free task, thereby improving resource utilization and minimizing application makespan. In brief, when idle CPU cores are adequate, task scheduling works like a regular scheduler. However, when inadequate, D2PTS will try to find an appropriate CPU core for the pending tasks that meets the following two conditions: (1) the task running on it has finished data transmission, namely in the network-free phase, (2) the data transmission time of the new task is larger than the remaining calculation time of the former task. The first condition is to ensure that when the new task reads remote data via the network (in network-intensive phase), the former task is performing calculations (in network-free phase). The second condition is to ensure that when the new task finishes data transmission, the former task has completed execution, so as to reduce competition for resources. In summary, the key issue to be addressed is how to calculate the data transmission time and the remaining calculation time.

4.1 Task Data Distribution Statistics

Task data volume is an important parameter for calculating the data transmission time and the remaining calculation time of tasks. In Spark, a stage is submitted only when all its upstream stages have been completed. Through this, we can accurately calculate the distribution of the data to be processed by the task when the stage it belongs to is submitted. For easier narration, we divide the stage into *InputStage* and *IntermediateStage*: *InputStage* reads input data from external data source, like Hadoop Distributed File System (HDFS) [31], while *IntermediateStage* typically reads input data using a network shuffle, where each task reads data blocks of output data from all of the upstream stage's tasks.

For *InputStage*, we leverage the lineage relationship to find the first RDD of the stage. This RDD is generated based on external data. Different data sources generate different types of RDD. Each type of RDD has a corresponding companion partition type. For example, HadoopRDD generated based on HDFS has a companion object HadoopPartition, from which we can obtain the information of data partition (e.g., data location, data length, etc.).

For *IntermediateStage*, its data comes from the parent stage through shuffle. Spark employs a unique id—*shuffleId* to identify the dependency between these two stages. Once the tasks in its parent stage complete executing, the location of the intermediate data they outputs with a *shuffleId* will be written to the MapOutputTrackerMaster component in driver. This means that the information of data partition processed by tasks in the downstream stage can be obtained through the following three steps: (1) calculating *shuffleId*; (2) employing *shuffleId* to get the intermediate data information of the corresponding shuffle phase from the MapOutPutTracker component; and (3) calculating the size of the data partition for every task. There are similar intermediate calculation results collection steps for different frameworks.

Algorithm 1 describes the calculation process of the data distribution and total data size for tasks, which is invoked when the stage is submitted. The amount of remote data of the task is relative to the host, and the amount of remote data that the task needs to fetch on different hosts is also different. Algorithm 2 describes how to calculate the size of data that a task needs to fetch when running on a specific host.

4.2 Data Transmission Time Estimation

The data transmission time is used to quantify the time a task costs in the network-intensive phase. For *InputStage*, if its task can be started on the node where the data is located, the task does not need to fetch data from the remote node when it runs. The data transmission time of the task that does not fetch data from the remote node is considered as 0. For another case, the data transmission time is calculated in the same way as the task in *IntermediateStage*. For the *IntermediateStage*, since the tasks of its parent stages run on multiple nodes, the intermediate data generated also spreads across multiple nodes. Therefore, when a task is executed, it first needs to fetch corresponding data from multiple nodes, and then performs a series of calculation operations on them before outputting results.

In a distributed computing system, many tasks are running at the same time, and the available resources in the cluster will change at any time. Therefore, it is very difficult and costly to calculate the precise data transmission time of the task by collecting real-time statistical information during data transmission. In response to this problem, we propose to use the task's data transmission process as a black box, make full use of the task's data transmission history information, and use machine learning technology to predict the approximate data transmission time of the task.

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1. Task Data Distribution Statistics

Input:
The stage to compute: $stage$.
Output:
The mapping of tasks and their total sizes: $taskSizes$;
The mapping of tasks and their remote size set: $taskRemoteSizes$.
// Initialize the output of the two mappings.
$taskSizes$ = ∅;
$taskRemoteSizes$ = ∅;
if $stage$ == *InputStage* **then**
  for *each* $task_i$ *in stage.tasks* **do**
    // Obtain directly from the storage location.
    $taskSizes$.put($task_i$.id, $task_i$.dataSize);
    // In the case of multiple copies, the partition exists on multiple hosts.
    $locs$ = $task_i$.getCopiesLocations();
    // Initialize the mapping of hostname and the size of data contained on it.
    dis = ∅;
    for *each* loc_i *in* $locs$ **do**
      $dis(loc_i)$ = $task_i$.dataSize;
    end
    $taskRemoteSizes$.put($task_i$.id, dis);
  end
end
if $stage$ == *IntermediateStage* **then**
  // Obtain *OutPutStatus* from the $stage$.
  OutPutStatus = $stage$.getIntermediateResult();
  // Start calculating the size of the data contained in each task
  for *each* $task_i$ *in stage.tasks* **do**
    // Initialize the total block size.
    $totalOutputSize$ = 0;
    // Initialize the mapping of hostname and the size of data contained on it.
    dis = ∅;
    // OutPutStatus contains multiple blocks.
    for *each* $status_i$ *in OutPutStatus* **do**
      // Get block size based on task id;
      $blockSize$ = $status_i$.getSizeForBlock($task_i$.id);
      if $blockSize > 0$ **then**
        $totalOutputSize$ += $blockSize$;
        $dis(status_i$.host) = dis.getOrElse($status_i$.host, 0L) + $blockSize$;
      end
    end
    $taskSizes$.put($task_i$.id, $totalOutputSize$);
    $taskRemoteSizes$.put($task_i$.id, dis);
  end
end
return ($taskSize$, $taskRemoteSizes$)

Since the task does not require the participation of the CPU during the data transmission phase, there is basically no situation in which CPU load fluctuations affects the data transmission time. The input data of the task can have multiple sources, including the memory of the node, and the remote node. Different sources represent the consumption of different resources, and the performance of the resources will affect the time of data transmission. After the above analysis, we can summarize the factors that affect the task data transmission time, including the size of the transmitted data, the number of simultaneous data transmission requests, and the network bandwidth. All these factors can be obtained after the task is generated.

Algorithm 2. Get Remote Data Size for Tasks

Input:
The mapping of tasks and their total sizes: $taskSizes$;
The mapping of tasks and their remote size set: $taskRemoteSizes$;
The id of task: $taskId$;
Host name: $hostname$.
Output:
The size of the remote data that needs to be fetched: $fetchSize$.
// Initialize the remote data size.
$fetchSize$ = 0;
// Calculate the data distribution.
dis = $taskRemoteSizes$.get($taskId$);
if $hostname$ *is in* dis **then**
  $fetchSize$ = $taskSizes$.get($taskId$) - dis.get($hostname$);
else
  $fetchSize$ = $taskSizes$.get($taskId$);
end
return $fetchSize$

Based on the factors that affect the data transmission time, the transmission time estimation can be formulated as a regression prediction problem. We design an algorithm based on back propagation (BP) neural network [32] for tasks that fetch remote data via a network to predict [33] the data transmission time. In an actual production environment, a large number of applications will be running online all the time, which creates an opportunity for us to plan ahead.

In order to verify the effectiveness of the machine learning model in predicting the approximate data transmission time and the superiority of the proposed BP neural network, we also use other classic machine learning models to solve regression problems as comparative experiments. We conduct experiments with a data set containing 134,000 rows of records, randomly selected 70% of the data set as the training set, and the remaining 30% as the test set, and use the Root Mean Square Error (RMSE) as the evaluation indicator of the model, as shown in the Eq. (1), where M represents the training data set, m represents each piece of training data, $\widetilde{y_m}$ represents the predicted value of the model, and y_m represents the actual value.

With the continuous iterative update of the model, the Root Mean Square Error on the test set and training set continues to shrink, which proves that the model's prediction accuracy for task data transmission time on the training set and test set continues to increase, which also proves the learning of the model is effective. The model can learn useful knowledge to predict the transmission time of new task data and can guarantee a certain degree of accuracy. The final experimental results of all models on the data set are shown in the following Table 1.

$$RMSE = \sqrt{\frac{1}{|M|} \times \sum_{m \in M} (\widetilde{y_m} - y_m)^2}. \quad (1)$$

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
RMSE of the Models

Support Vector Regression	6.856
Multiple Linear Regression	5.041
Decision Tree Regression	5.785
Random Forest Regression	5.108
Extreme Gradient Boosting	4.836
BP neural network	4.569

From the experimental results, the machine learning models proposed above have achieved good results, which proves the effectiveness of feature extraction. Among them, BP neural network achieves the best effect on the data set, reducing the RMSE to 4.569, which proves the superiority of the model for nonlinear fitting on a data set with a small number of features. Therefore, this paper chooses BP neural network as a model for predicting the approximate time of task data transmission.

For better description, some of the parameters used are shown in Table 2.

The architecture of our BP neural network used in D2PTS is illustrated in Fig. 5a, in this study, it is a four-layer network , consisting of input layer, two hidden layers and output layer. We present the train set as a collection of task vectors: $V = [S_i, R_i, N_i, Tran_i]$. The first three elements in the task vector are corresponding parameters to affect data transmission time. During the job runtime, the Recorder in the worker node records this information and reports it to Collector, which collects and stores it. This data will then be used in the model training process. The activation function is as follows:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}. \tag{2}$$

The loss function f uses mean square error:

$$f = \frac{1}{n} \times \sum_{t=1}^{n} (\widetilde{y_t} - y_t)^2. \tag{3}$$

The overall process of predicting the task's data transmission time is shown in Fig. 5b, it leverages historical job information for offline training. During the actual D2PTS runtime, the online predictor receives the characteristics of the tasks (e.g., remote input data size, number of requests, network state) and then uses these characteristics to estimate data transmission time of tasks. For reasons of Spark, these characteristics of the tasks are not directly available, we need to do extra works to calculate the following three key values:

TABLE 2
Variable Declaration

S_i	the remote input data size of the ith task;
R_i	the number of data fetch requests of the ith task;
N_i	the average network speed;
$Tran_i$	the data transmission time of the ith task;
f	the loss function;
n	the number of training data of the ith task;
$\widetilde{y_t}$	the predictive of value the ith task;
y_t	the actual value of value the ith task;

- remote input data size. The entire calculation is shown in the procedure to work out the parameter $taskRemoteSizes$ in Section 4.1. We can obtain the remote data size of a particular task by using its $taskId$ and the hostname of the host it will run on.
- the number of data fetch requests. Tasks fetch data from remote hosts by sending requests, so the number of requests affects the transmission time of remote data.
- network state. It refers to the network status of the node to which the task will be assigned. We use Linux netstat to monitor the network status and implemented a distributed monitoring tool (Recorder) to receive network status.

After the task actually finished fetching the remote data, the Recorder reports the above three parameters together with the actual data transmission time to Collector to form a training set.

Over time, the size of stored tasks information has increased dramatically. To our knowledge, if the training set is too small, the training accuracy will be insufficient. If the training set is too large, the training time will be too long. In order to effectively solve this conflict, we use the data of the past 10 days as the training set, and take offline training once a day. As shown in Fig. 5c, if the training interval is one day, the training set represents the data for the past 10 days. The two parameters above are configurable for user.

The actual data transmission time may be affected by other ongoing data fetch tasks, we tradeoff accurate (absolute) transmission time for simpler and practical estimator. But in the experimental evaluation, we can see that the approximate time estimation can achieve a significant performance improvement.

4.3 Remaining Calculation Time Estimation

We employ task calculation time (not including remote data fetch time) to quantify the time a task is in the network-free phase. In a Spark framework, tasks in the same stage perform

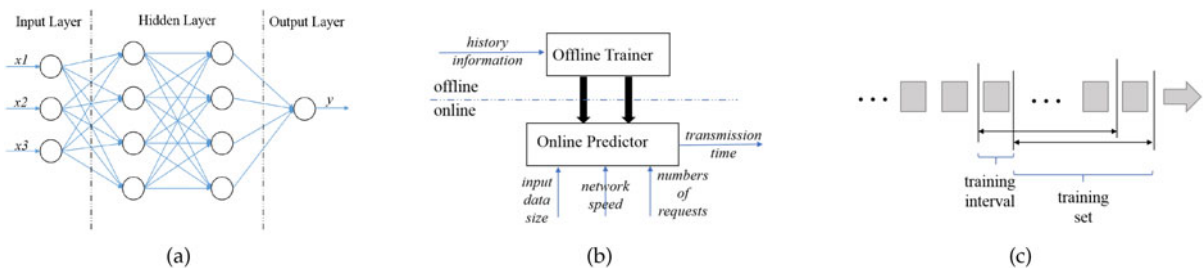


Fig. 5. Data transmission time estimation.

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Variable Declaration

$taskId$	the id of the task;
$currentTime$	current system time;
D_f	the data size of the finished task;
T_f	the calculation time of the finished task;
v	processing speed of executor;
D_i	the data size of the ith task;
T_i	the calculation time of the ith task;
T_i^{curr}	current system time;
T_i^{tran}	the finishing data transmission time of the ith task;
T_i^r	the remaining calculation time of the ith task;

the same calculations using different subsets of the data set. In our study, using the calculation time of completed tasks to estimate the calculation time of executing tasks in the same stage is a natural way. After a task is completed, we mark its stage as an available stage, which means that the calculation time of other tasks in this stage can be estimated. The information of completed tasks, including the size of the data it processed, and the calculation time it was in network-free phase can be used to estimate the calculation time of other tasks that are being executed. We only calculate the time of the tasks when it is in the network-free phase, not the entire running time.

For better description, some of the parameters used are shown in Table 3.

Executor will send a signal in the form of $<taskId, currentTime>$ to Collector when the remote data fetch of the task is finished. A signal is also sent to Collector at the end of task execution. The purpose of estimating the remaining calculation time is to obtain the time that the task will still be in the network-free phase.

Utilizing these task characteristics above, which can be obtained from previous task runs, we formulate a simple model to calculate the remaining calculation time of tasks. Assuming all executors process the task at a similarly constant speed, the remaining calculation time of a task i being executed can be estimated using Eq. (4).

In general, the computing node of the executor always own multiple CPU core, and multiple tasks can be executed in parallel on multiple cores. The remaining calculation time of certain executor can be calculated by the longest one amongst of the remaining time of all tasks running on the same executor. The remaining calculation time of the jth executor can be calculated from Eq. (5).

$$\begin{cases} v = \frac{D_f}{T_f} \\ T_i = \frac{D_i}{v} \\ T_i^r = T_i - (T_i^{curr} - T_i^{tran}) \end{cases} \tag{4}$$

$$T_j = max_{i \in E_j} T_i^r, \tag{5}$$

where i represents the id of the tasks that is running on E_j and is in the network-free phase.

If the remaining calculation time of a certain executor is shorter than the transmission time of the task to be scheduled, D2PTS will reserve these two tasks for the same CPU core to take advantages of high resource utilization. After the task with the shortest remaining time already has an *Advance* task, other tasks will still determine whether they can coexist with the task on the executor. This makes the actual remaining calculation time of a certain executor different from the estimation results. In fact, the remaining calculation time of the executor is not the minimum remaining time of the previous task. To avoid this situation, once a task has become an *Advance* task of the task with the shortest remaining time, we label the running task, and exclude the labeled task while calculating remaining computing time for other tasks next time.

4.4 Selecting Advance Tasks

When submitting the stage, Spark will set the preferred execution position for each task of the stage according to the data distribution to be processed by the task on each node. At the same time, Spark maintains a series of preferred queues. Different preferred queues of task represent different levels of data locality. Spark defines five levels of data locality for tasks, from high to low: $PROCESS\_LOCAL$, $NODE\_LOCAL$, NO_PREF, $RACK\_LOCAL$ and ANY. Scheduling for a certain level of data locality will find tasks to be scheduled from the corresponding task queue. Typically, we want tasks to be performed with high data locality.

In order to adapt our proposed scheduling algorithm to other frameworks without a data locality mechanism, we hide the data locality mechanism of Spark when implementing the algorithm, and abstract the input of the scheduling algorithm into a list of tasks to be scheduled to achieve the goals that algorithm does not depend on a specific framework.

In Spark, each worker node of the cluster will launch multiple executor processes and run multiple tasks in a multi-threaded manner. Task scheduling is triggered by the advent of idle resources. Whenever the executor resource is idle, Spark's task scheduler will select the appropriate tasks to schedule to that executor to run or wait for the arrival of the more appropriate idle executor. According to the definition of D2PTS, it seems that the most straightforward implementation is to calculate the data transmission time of all tasks in the preferred list, and monitor all the running tasks. Once a new task that has a data transmission time greater than the remaining calculation time of any running task is found, the Scheduler of D2PTS schedules this task to the executor where the former task is.

In order to solve the above problem, our model is triggered at the end of task execution and it is compatible and cost-saving with FIFO scheduling. In FIFO task schedule mechanism, when a task completes its execution, the corresponding CPU core is considered idle, which then triggers a resource offer for the executor where the task is located. In the D2PTS model, because there may be two tasks located on the same CPU core, the completion of tasks does not mean that the resources must be idle. When a task sharing a CPU core with an *Advance* task completes execution, if a new task is scheduled directly, it will exacerbate the burden on the cluster and cause performance degradation. To prevent this from happening, when the task that completes execution is a task that has an *Advance* task, it blocks the executor's message to the driver about the status completion of the task. Therefore, only our D2PTS module will be called to find *Advance* tasks. The selection process of *Advance* tasks can be divided into the following steps:

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
Configuration of Cluster

Type	Configuration
Number of Nodes	1 master, 8 workers
Memory(GB)	32
CPU Number	4
Network	1000Mb/s Ethernet
Environment	Ubuntu 16.04, Spark 2.4.3,Hadoop 2.6.0

TABLE 5
Configuration of Spark Parameters

Type	Configuration
CPUS_PER_TASK	1
Executor Cores	4
Number of executors Per Worker	1
Number of advance tasks	4

1) Calculate data transmission time. Calculate the data transmission time $Tran_i$ of the ith task in the task list to be scheduled.
2) Sort tasks. Sort the tasks in descending order according to the data transmission time.
3) Calculate remaining calculation time. Calculate the remaining calculation time T_j of the executor j where the task finished.
4) Find *Advance* tasks. Find a task with the max $Tran_i$ and satisfy $Tran_i$ greater than T_j in the sorted task list. If it exists, it means that the task is *Advance* task can be scheduled in advance. If it doesn't exist, it means that there is no *Advance* task, and the scheduling ends.
5) Return and schedule the *Advance* task.

The scheduling process of D2PTS is triggered by the signal sent by the Recorder component of D2PTS. When the D2PTS Scheduler receives the signal that the task execution is complete, it will first determine whether there is an Advance task on this resource. If it does not exist, then it means that the resource is completely idle, then use the FIFO strategy to schedule a task to the resource from the task list to be executed. If there is an Advance task on this resource, this scheduling ends. When the Scheduler receives a signal from the Recorder that the data transmission of a certain task is completed, it will enter the process of finding and scheduling Advance tasks. Specifically, the Scheduler first estimates the data transmission time of each task in the list of tasks to be scheduled, and sorts it in descending order. Then Scheduler calculates the remaining calculation time of the task that completed the data transmission, and finds a task with the largest data transmission time in the sorted task list, and the data transmission time is greater than the remaining calculation time. This is the advance task that meets the requirements. It is scheduled to the resource corresponding to the task that completed the data transmission.

Later, we will verify through experiments: D2PTS only needs to perform simple calculations in the case of CPU resource occupation, and has almost no impact on scheduling performance. And the pipeline task execution model improves the utilization of resources, reduces the waiting time of tasks, and thus reduces the execution time of the entire job. Algorithm 3 describes the entire D2PTS workflow.

Algorithm 3. Dual-Phase Pipeline Scheduling Model

Input:
The newly completed task and its executor: ($task_i, executor_j$),
The list of tasks to be scheduled: $taskList$.
if $hasAdvanceTask(task_i) == false$ **then**
  // $executor_j$ does have idle CPU resources.
  // Schedule task with FIFO policy.
  t = $taskList$.getFIFOTask();
  Schedule new task t to $executor_j$;
end
// Receive data transmission complete signal
if *receivedSignal()* **then**
  // Initialize the mapping of tasks and its transfer time.
  $M = \emptyset$;
  for *each* $task_i$ *in* $taskList$ **do**
    $Tran_i$ = calculateTransferTime($task_i$);
    $M.put(task_i, Tran_i)$;
  end
  // Sort elements in M by transmission time.
  M.sort();
  //Calculate the remaining computing time of $executor_j$.
  $rTime$ = calculateRemainingTime($executor_j$);
  for *each pair* $(task_k, Tran_k)$ *in* M **do**
    // If the data transmission time is greater than the remaining calculation time.
    // Ensure that AdvanceTask will not compete for CPU with existing task.
    if $Tran_k > rTime$ **then**
      $AdvanceTask$ = $task_k$;
      $taskList$.remove($task_k$);
      berak;
    else
      break;
    end
  end
  Schedule the $AdvanceTask$ to $executor_j$;
end

5 Experiments

5.1 Experiment Setting

By modifying the core module of the Spark source code, adding components as described in Section 3, our D2PTS is embedded into the original Spark framework. In this section, to verify the performance of D2PTS, we conduct a lot of experiments on a cluster based on Spark 2.4.3 with 1 master node and 8 worker nodes. The hardware and software configurations are shown in Table 4. The deployment mode is Spark on Standalone and the Spark configurations are shown in Table 5.

In these experiments, the scheduling methods we used for comparison are as follows:

- Fair-Spark: It represents Spark's common scheduling policy.
- Symbiosis ([23]): It is a network-aware task scheduler used in data-parallel frameworks (such as Spark).

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 6
Benchmark Workloads

Application types	Benchmarks
search engine applications	PageRank
map side combination application	WordCount
simple applications	Sort

Whenever task scheduler launches a network-intensive task, Symbiosis will fill the same executor with a computation-intensive from the pending tasks at the same time.

The following indicators are used for experimental performance comparison:

- Job execution time: The time from when a job is executed to when it is finished. It is a universal and intuitive performance comparison indicator.
- Resource utilization: D2PTS mainly improves cluster performance from the perspective of improving resource utilization, including CPU and network resources.

5.2 Workload Detail Description

To comprehensively evaluate performance of D2PTS, as shown in Table 6, one multi-stage application ($PageRank$) and two two-stage applications ($WordCount$, $Sort$) are selected from HiBench [34]. We use $WordCount$ to test the performance on the application with less data transmission in the case of map side combination, but use $Sort$ in the opposite case. $PageRank$ is used to evaluate the performance on multi-stage Iterative application.

The following is a detailed description of the three workloads:

• $WordCount$ This workload contains two stages. The first stage is called MapStage. Its tasks are responsible for pulling data that needs to be calculated from external data sources, such as local files, hdfs, etc., during the network transmission stage. In the calculation phase, the data row is divided into words according to spaces and a count value 1 is added to each word to form a key-value pair <word, 1>. After that, each task accumulates its own internal key-value pairs with the same key, and then output to the intermediate file. The second stage becomes ReduceStage, and its task is responsible for pulling the output file of the previous stage from each node in the network transmission stage. At this time, each key only exists in one of the tasks, and the task also aggregates the same key. The key-value pair and the count value are accumulated to get the final count value of each word, and finally output to the result file.

• $PageRank$ This is a workload of multiple iterative calculations, and each iteration calculation will generate a stage. For the first stage, in the data transmission stage, the task is also responsible for pulling data from the external data source. In the calculation stage, the task calculates the contribution value of each page to its adjacent pages and outputs it to the intermediate file. For each subsequent stage, in the network transmission stage, the task pulls the intermediate results output by the previous stage from each remote node. In the calculation stage, the task aggregates and calculates the contribution value of each page and updates the calculation result to the sort value of each page, and repeat the calculation steps of the first stage. The page sort value after running the specified number of iterations is the final result. In this paper, 12 iterations are selected for the experiment.

• $Sort$ This workload contains two stages. In the first stage, the task of the network transmission stage is still to pull the specified data from the external data source. In the calculation stage, the task first estimates the distribution of the keys through sampling, and then divides the boundaries of each range according to the specified order, and recalculate the partition to which each key belongs according to the boundary of the range. After this stage, it can be ensured that the data between the partitions corresponding to each range is sorted according to the partition index. In the second stage, each task gets its own partition data and sorts the partitions. After these stages are completed, the sorting between partitions and within the partitions has been completed, and the result data is obtained in the order of the partition index and combined to obtain the ordered results after the global sorting.

5.3 Performance Results

To verify the performance of D2PTS, we perform a series of experiments on the different datasets using the benchmarks from HiBench in Table 6. Each benchmark was performed 10 times, and the results were averaged as the final result. In this experiment, D2PTS is compared with Spark's Fair scheduling strategy and Symbiosis scheduling strategy.

5.3.1 BP Neural Network Performance

To show the effect and potential of the BP neural network model more clearly, we used models of different accuracy to conduct experiments on three benchmarks and collected data on various performance indicators. The experimental results are shown in Fig. 6. Obviously, for all benchmarks, as the accuracy of the model increases, multiple performances will increase accordingly. This is because the model's accuracy affects the prediction accuracy of the advance task data transmission time during D2PTS scheduling. If the precision is too low, it may cause competition between the scheduled advance task and the task currently being calculated by the CPU, resulting in a decrease in performance. In subsequent experiments, the model's accuracy we used to estimate the data transmission time was 95%.

5.3.2 D2PTS versus Fair-Spark

The experimental results are shown in Fig. 7. Obviously, by comparing the job completion time in Fig. 7, D2PTS has improved performance compared to the Fair scheduling strategy. In Fig. 7, we show job completion times for three benchmarks, $PageRank$, $WordCount$, and $Sort$. Performance boost varies between 8.76% and 13.06%. In particular, $PageRank$ jobs benefit more from D2PTS because of more stages they have.

More specifically, $WordCount$ is a widely used application for counting the frequency of words in the field of data processing. We counted job completion times on three different datasets: 10G, 20G and 40G. D2PTS works for this

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

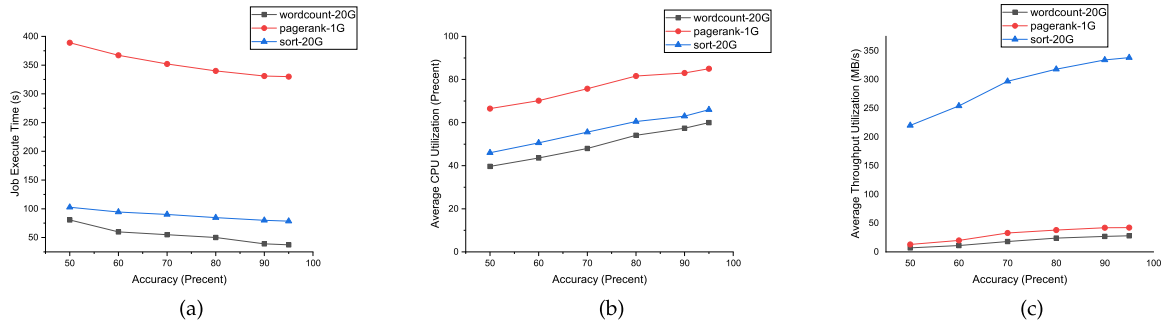


Fig. 6. Comparisons of program performance under different model accuracy.

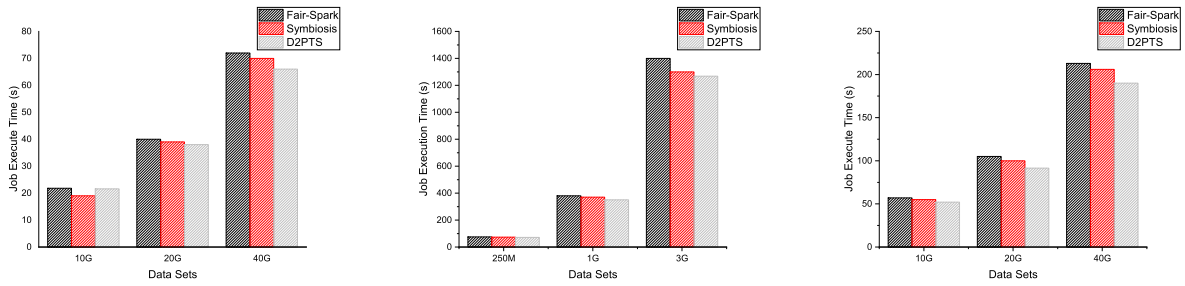


Fig. 7. Comparisons of Fair-Spark, Symbiosis and D2PTS under different benchmarks.

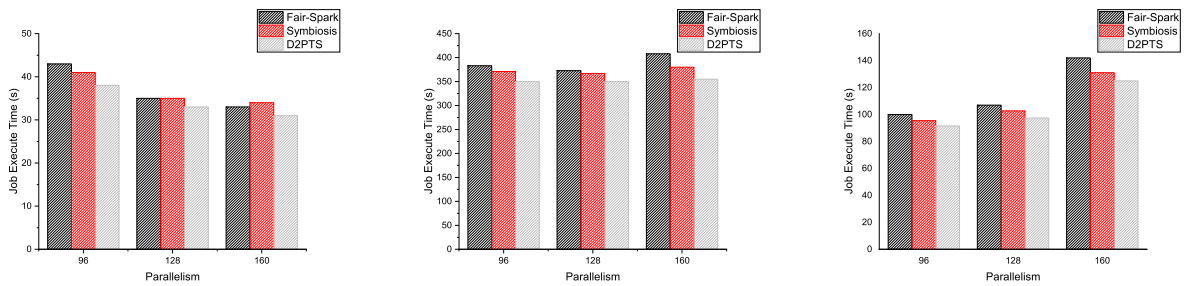


Fig. 8. Comparisons of Fair-Spark, Symbiosis and D2PTS under different parallelism.

type of application mainly because there exist tasks that require data transmission during the *Map* phase. In this program we use the *reducebyKey* operator, which will perform a map side combination, so less data is transmitted. Therefore, D2PTS has less effect on the *Reduce* stage. From Fig. 7a, we can see that as the amount of data increases, D2PTS works better. This is because the larger the amount of data, the greater the number of data storage blocks on HDFS and the greater the number of tasks in the *Map* phase. Similarly, *Sort* is also an application of two stages but there is a large amount of data transmission during the shuffle phase. Experimental results prove that D2PTS works better in *Sort* applications than *WordCount* applications. The average performance gain of *Sort* applications is 11.4% while *WordCount* is 8.37%.

PageRank is an iterative search engine algorithm, and the number of stages (N_{stage}) included in such applications is related to the number of iterations ($N_{iteration}$), which satisfies $N_{stage} = N_{iteration} + 3$. Unlike the other two applications, *PageRank* has multiple stages. Therefore, the experimental results of Fig. 7 are obtained with 10 iterations. From the Fig. 7b, when the amount of data is small, the gap between D2PTS and Fair-Spark is insignificant. However, as the amount of data increases, the data transmitted during the shuffle phase also increases, and D2PTS works better. Also, we performed a lot of experiments under different iterations. As shown in Fig. 9, as the number of iterations increases, the number of stages increases. Although the total application program time is increasing, because D2PTS works in each stage, the time magnitude of D2PTS reduction is also increasing.

In order to better observe the effect of the number of tasks on D2PTS, we conduct a series of experiments with different degrees of parallelism, and the experimental results are shown in Fig. 8. Obviously, as the degree of parallelism increases, the gap between Fair-Spark and D2PTS also increases, because D2PTS can schedule tasks for execution in advance in the case of multiple batches of tasks. The degree of parallelism is an artificially configured parameter. Excessive parallelism results in a lot of overhead for thread switching while if parallelism is too small, there is a long idle period of the node. It is unwise to expect that every user can configure the parallelism reasonably. D2PTS can mitigate the impact of unreasonable configuration.

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

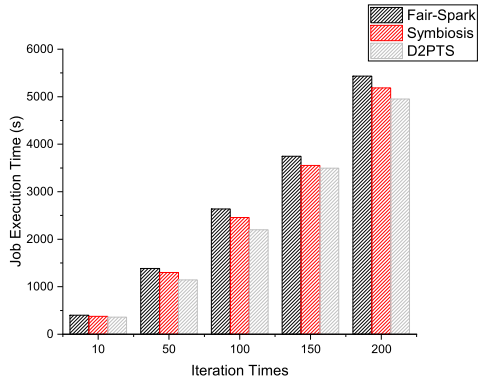


Fig. 9. Pagerank benchmark on different number of iterations.

5.3.3 D2PTS versus Symbiosis

Symbiosis is a typical online scheduler that co-locates computation-bound and network-bound tasks in the same executor process. Similarly, D2PTS and Symbiosis both do not explicitly increase the number of executor cores, but instead schedule eligible additional tasks. Symbiosis distinguishes tasks in a way where data and task are in the same process or node as compute-bound, and the otherwise as network-bound.

The results of the experiment are also shown in Figs. 7 and 8. Symbiosis has improved performance compared to a fair scheduling strategy. This is because Symbiosis increases the number of tasks running simultaneously on each executor. But compared with others, D2PTS has higher performance in most cases. This is because Symbiosis does not consider the different resources required for each stage of the task when scheduling, but simply schedules the tasks together for execution, which will bring competition for resources and cause performance degradation.

For the *WordCount* workload with a 10G data set size, Symbiosis is better than D2PTS because the data set is relatively small, so the network bandwidth can withstand multiple tasks to transmit data at the same time, and it also saves the cost that D2PTS takes to estimate and calculate. Although resource competition will occur during the simultaneous execution of multiple tasks, the parallelism of the executor is improved within an acceptable range and the scheduling cost is reduced, so the overall time is reduced. For *PageRank* workload, in Symbiosis, most tasks in *IntermediateStage* perform data transfer so that they cannot be launched in advance as a symbiotic task. As shown in Fig. 9, in *PageRank* workload with different numbers of iterations, the performance of D2PTS is also better than Symbiosis. Moreover, if we launch two tasks at the same time, serious resource competition will occur. D2PTS avoids competition for resources by staggering time to achieve better performance. All in all, D2PTS is a more general model and performs better in most cases.

5.3.4 Effect of Resource Utilization

To present resource utilization, we sample the resource utilization including CPU and network over a period while jobs are running. Figs. 10 and 11 shows the runtime resource utilization of the above three types of jobs. D2PTS significantly improves CPU utilization and network throughput in all the three workloads. Compared to the Fair scheduling strategy and Symbiosis that simply places tasks into available slots but neglects the fact that the resource requirements of tasks are dynamically changing, D2PTS performs tasks in a coarse-grained dual-phase pipeline manner, improving CPU utilization and network throughput. Specifically, the CPU utilization rate increased from 18.1% to 46.6%, and the throughput utilization rate increased from 20.2% to 36%. When an executing task is in the network-free phase, D2PTS additionally schedules a new network-intensive task at the right time, which avoids the CPU idle period of a new task while fetching

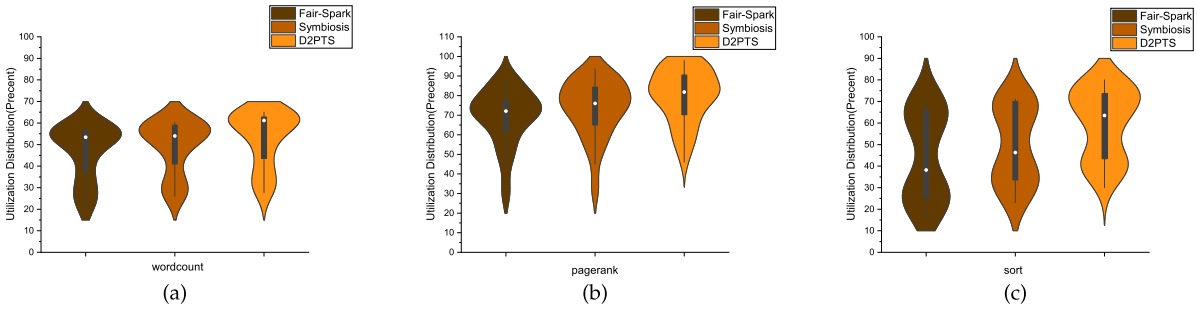


Fig. 10. CPU utilization.

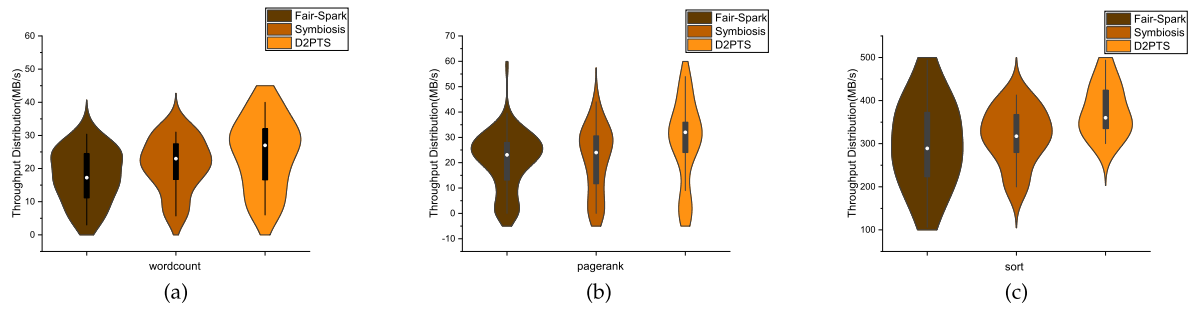


Fig. 11. Network throughput utilization.

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 7
Scheduling Overhead Statistics

Scheduler	Workload	Scheduling Time(s)	Percentage
Fair-Spark	WordCount	5.22	14.1%
	PageRank	67.12	17.3%
	Sort	14.07	13.4%
Symbiosis	WordCount	3.03	8.4%
	PageRank	34.4	9.2%
	Sort	8.78	8.7%
D2PTS	WordCount	3.88	11.1%
	PageRank	46.12	12.6%
	Sort	8.83	9.7%

remote data. D2PTS adopts a "coax" approach to make the executor run more tasks than the original setup in order to make fuller use of resources. In addition, we find that *Sort* applications have the greatest improvement in CPU resource utilization among the three. Because such applications tend to have larger network data transmissions, there is a longer quiet period for CPU resources.

5.3.5 Scheduling Overhead Comparison

In order to verify the effectiveness of D2PTS more comprehensively, we collect the time consumed by three different scheduling strategies for processing scheduling logic under three workloads, and calculate the proportion of the job execution time. The results of the experiment are shown in the Table 7 below.

For Fair-Spark, it belongs to the input-compute-output mode, so the scheduling of the next task is triggered when the previous task is completed. We also count the idle time between these two task scheduling into the scheduling overhead. For Symbiosis, it simply divides the tasks into two categories to perform overlapping scheduling and execution. Because the impact of the duration of the task is not considered, there is no more calculation, and the cost of scheduling is the lowest. But too simple overlapping execution will bring resource conflicts and cause performance degradation. For D2PTS, more fine-grained considerations are carried out compared to Symbiosis. In the scheduling process, the remaining calculation time of the task is calculated, the data transmission time is estimated and the relationship between the calculation phase and the transmission duration is estimated. Although this slightly increases some scheduling costs, it also avoids resource idleness and competition.

5.4 Experiment Summary

In this section, we perform performance evaluations of D2PTS on different popular benchmarks in different scenarios. Based on the experimental results, we summarize as follows:

- D2PTS reduces application execution time by scheduling tasks in advance in the case of multiple batches of tasks with little overhead.
- D2PTS improves resource utilization by launching more tasks than the original setup. The number of tasks that each executor can run simultaneously is artificially configured, and D2PTS can mitigate the impact of unreasonable configuration.

6 Conclusion

In the parallel distributed data processing frameworks, their task scheduling mechanisms are likely to lead to underutilization of resources. A CPU core is not always fully used despite a task being assigned to it. This will cause the job to run longer. This paper attempts to improve the utilization of resources and make the job run faster. First, it estimates the remote data transmission time of the task to be scheduled and the remaining execution time of the executing task. Second, the task to be scheduled is scheduled in advance through effective heuristic rules to make up for the insufficient utilization of resources. When an executing task is in the network-free phase, D2PTS additionally schedules a new network-intensive task at the right time. These two tasks sharing the same CPU core are executed as a coarse-grained pipeline. This execution method can start tasks earlier and improve resource utilization.

We experiment on three benchmarks with different amounts of data and situations. The experimental results show that D2PTS can effectively reduce the completion time of the job and improve the utilization of resources with light overhead. In the case of unreasonable parallelism configuration, D2PTS still works well.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[2] "Hadoop," 2021. [Online]. Available: http://hadoop.apache.org/
[3] "Spark," 2021. [Online]. Available: http://spark.apache.org/
[4] "flink," 2021. [Online]. Available: https://flink.apache.org/
[5] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, 2012, pp. 15–28.
[6] M. R. Garey and D. S. Johnson, "Computers and intractability: A guide to the theory of NP - completeness," in *Mathematics of Operations Research*, New York, NY, USA: W. H. Freeman & Co., 1988.
[7] S. Wang, W. Chen, X. Zhou, L. Zhang, and Y. Wang, "Dependency-aware network adaptive scheduling of data-intensive parallel jobs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 3, pp. 515–529, Mar. 2019.
[8] Z. Li, H. Shen, and A. Sarker, "A network-aware scheduler in data-parallel clusters for high performance," in *Proc. 18th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2018, pp. 1–10.
[9] Z. Tang, W. Ma, K. Li, and K. Li, "A data skew oriented reduce placement algorithm based on sampling," *IEEE Trans. Cloud Comput.*, vol. 8, no. 4, pp. 1149–1161, Fourth Quarter 2016.
[10] M. Hammoud and M. F. Sakr, "Locality-aware reduce task scheduling for mapreduce," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, 2011, pp. 570–576.
[11] O. Selvitopi, G. V. Demirci, A. Turk, and C. Aykanat, "Locality-aware and load-balanced static task scheduling for mapreduce," *Future Gener. Comput. Syst.*, vol. 90, pp. 49–61, 2019.
[12] C. Navasca *et al.*, "Gerenuk: Thin computation over big native data using speculative program transformation," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 538–553.
[13] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[14] X. Li, F. Ren, and B. Yang, "Modeling and analyzing the performance of high-speed packet I/O," *Tsinghua Sci. Technol.*, vol. 26, no. 4, pp. 426–439, 2021.
[15] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 407–420, 2015.
[16] G. Ananthanarayanan *et al.*, "Scarlett: Coping with skewed content popularity in mapreduce clusters," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 287–300.

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.

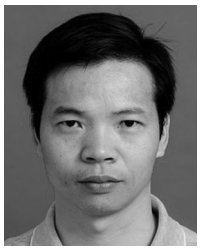
[17] Q. Chen, C. Liu, and Z. Xiao, "Improving mapreduce performance using smart speculative execution strategy," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 954–967, Apr. 2014.
[18] E. B. Nightingale, P. M. Chen, and J. Flinn, "Speculative execution in a distributed file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 191–205, 2005.
[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, 2008, Art. no. 7.
[20] Y. Mao, H. Qi, P. Ping, and X. Li, "FiGMR: A fine-grained mapreduce scheduler in the heterogeneous cloud," in *Proc. IEEE Int. Conf. Inf. Autom.*, 2016, pp. 1956–1963.
[21] J. Polo *et al.*, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2011, pp. 187–207.
[22] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.
[23] J. Jiang, S. Ma, B. Li, and B. Li, "Symbiosis: Network-aware task scheduling in data-parallel frameworks," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[24] P. Dhawalia, S. Kailasam, and D. Janakiram, "Chisel: A resource savvy approach for handling skew in mapreduce applications," in *Proc. IEEE 6th Int. Conf. Cloud Comput.*, 2013, pp. 652–660.
[25] J. Tan, X. Meng, and L. Zhang, "Coupling task progress for mapreduce resource-aware scheduling," in *Proc. IEEE INFOCOM*, 2013, pp. 1618–1626.
[26] Z. Li, Y. Zhang, Y. Zhao, Y. Peng, and D. Li, "Best effort task scheduling for data parallel jobs," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 555–556.
[27] J. Bae *et al.*, "Jointly optimizing task granularity and concurrency for in-memory mapreduce frameworks," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 130–140.
[28] G. Xu, C.-Z. Xu, and S. Jiang, "Prophet: Scheduling executors with time-varying resource demands on data-parallel computation frameworks," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2016, pp. 45–54.
[29] "mesos," 2020. [Online]. Available: http://mesos.apache.org/
[30] "yarn," 2021. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[31] "hdfs," 2021. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html
[32] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.
[33] Y. Jin, W. Guo, and Y. Zhang, "A time-aware dynamic service quality prediction approach for services," *Tsinghua Sci. Technol.*, vol. 25, no. 2, pp. 227–238, 2020.
[34] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops*, 2010, pp. 41–51.
[35] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 13–24.
[36] M. Armbrust *et al.*, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.

Zhuo Tang (Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China. He is currently a professor with the College of Information Science and Engineering, Hunan University. He is also the chief engineer of the National Supercomputing Center in Changsha. His majors are distributed computing system, cloud computing, and parallel processing for big data, including distributed machine learning, security model, parallel algorithms, and resources scheduling and management in these areas. He has published almost 100 journal articles and book chapters. He is a member of ACM and CCF.

Zhanfei Xiao received the bachelor's degree in information management and information system from Guangxi University, China. He is currently working toward the master's degree at the College of Information Science and Engineering, Hunan University, China. His research interests include the parallel computing, big data parallel processing, distributed system architecture, especially the improvement and optimization of Spark framework.

Li Yang received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2013. She is currently an assistant professor with the College of Electrical and Information Engineering, Hunan University. Her research interests include distributed information systems, bioinformatics, artificial intelligence, and biomedical text mining.

Kailin He received the bachelor's degree in software engineering from Hunan Normal University, China. She is currently working toward the master's degree at the College of Information Science and Engineering, Hunan University, China. Her research interests include the parallel computing, big data parallel processing, distributed system architecture, especially the improvement and optimization of MapReduce framework.



Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He is currently a Cheung Kong professor of computer science and technology with Hunan University, the assistant to the President of Hunan University, the dean of the College of Information Science and Engineering, Hunan University, and the director with the National Supercomputing Center in Changsha. His major research interests include parallel and distributed processing, high-performance computing, and big data management. He has published more than 320 research papers in international conferences and journals such as *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Cloud Computing*, AAAI, DAC, SC, ICPP, etc. He is an Distinguished Member of the CCF. He is currently serving or has served as an associate editor for *IEEE Transactions on Computers*, *IEEE Transactions on Industrial Informatics*, and *IEEE Transactions on Sustainable Computing*.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: BEIJING UNIVERSITY OF POST AND TELECOM. Downloaded on December 19,2024 at 08:24:26 UTC from IEEE Xplore. Restrictions apply.