TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 05/10 pp487–497
DOI: 10.26599/TST.2019.9010054
Volume 25, Number 4, August 2020

Balance Resource Allocation for Spark Jobs Based on Prediction of the Optimal Resource

Zhiyao Hu, Dongsheng Li*, and Deke Guo

Abstract: Apache Spark provides a well-known MapReduce computing framework, aiming to fast-process big data analytics in data-parallel manners. With this platform, large input data are divided into data partitions. Each data partition is processed by multiple computation tasks concurrently. Outputs of these computation tasks are transferred among multiple computers via the network. However, such a distributed computing framework suffers from system overheads, inevitably caused by communication and disk I/O operations. System overheads take up a large proportion of the Job Completion Time (JCT). We observed that excessive computational resources incurs considerable system overheads, prolonging the JCT. The over-allocation of individual jobs not only prolongs their own JCTs, but also likely makes other jobs suffer from under-allocation. Thus, the average JCT is suboptimal, too. To address this problem, we propose a prediction model to estimate the changing JCT of a single Spark job. With the support of the prediction method, we designed a heuristic algorithm to balance the resource allocation of multiple Spark jobs, aiming to minimize the average JCT in multiple-job cases. We implemented the prediction model and resource allocation method in ReB, a Resource-Balancer based on Apache Spark. Experimental results showed that ReB significantly outperformed the traditional max-min fairness and shortest-job-optimal methods. The average JCT was decreased by around 10%–30% compared to the existing solutions.

Key words: Spark jobs; resource over-allocation; performance prediction

1 Introduction

Big data analytics is challenging for a single computer because large input data may result in an out-of-memory problem. The well-known MapReduce framework aims to partition large input data, which are processed by multiple computers concurrently. Apache Spark is a well-known in-memory MapReduce computing system, which is deployed on multiple computers. These computers provide computational resources and are interconnected via a computer network. The entire system of computers and network is called a "Spark cluster". Users submit their big data analytics as Spark jobs. The Spark cluster allocates computational resources to execute Spark jobs.

For Spark jobs, the amount of allocated resource is customized by the user, which significantly impacts the Job Completion Time (JCT). When computational resources are over-allocated, considerable computational tasks are executed concurrently. Hence, the network communication and disk I/O operations are, more frequently, causing larger system overheads. It has been reported that such system overhead takes up over 50% of the JCT[1]. We measured the JCT of Logistic Regression (LR) when the number of CPU cores increased from 30 to 100. Figure 1 shows the detailed JCT, including the

• Zhiyao Hu and Dongsheng Li are with the College of Computer, National University of Defense Technology, Changsha 410073, China. E-mail: huzhiyao14@nudt.edu.cn; dsli@nudt.edu.cn.
• Deke Guo is with the College of System Engineering, National University of Defense Technology, Changsha 410073, China. E-mail: guodeke@gmail.com.
∗ To whom correspondence should be addressed.
Manuscript received: 2019-03-30; revised: 2019-07-24; accepted: 2019-09-09

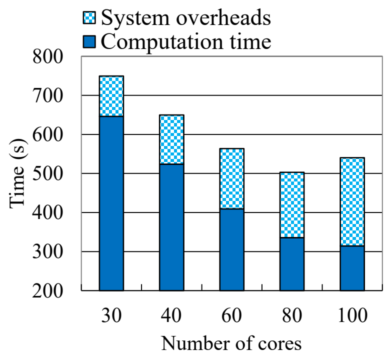


Fig. 1 JCT of logistic regression when the number of cores increases from 30 to 100.

system overhead and computation time. Its computation time decreased, but system overheads increased. In summary, the JCT decreased first and then increased.

Previous research has also observed that communication overhead is very large for distributed computation[2,3]. For Spark jobs, data shuffle and the storage of intermediate results incur considerable communication and disk I/O overheads[4]. For a single Spark job, the amount of allocated computational resources determines the number of tasks that are executed concurrently. However, determining the optimal allocation, specifically the over-allocation of a Spark job, remains an unsolved problem. Previous prediction methods have adopted high-level job settings, such as the size of input data and the number of virtual machines[5,6]. However, these methods have not adequately considered system overheads incurred by communication and disk I/O operations. To address this issue, we designed a Deep Neural Network (DNN)-based prediction mode.

In addition to the optimal allocation of a single Spark job, we present the resource allocation problem of multiple Spark jobs. In multiple-job cases, cluster resource with a finite capacity may not meet the optimal allocation of all jobs. Intuitively, allocation of the shortest job should be allotted adequate resources. However, their JCTs would not be optimized remarkably. Moreover, the shortest job may consume considerable computational resources, causing other jobs to suffer from under-allocation, and, finally a prolonged average JCT. Motivated by this fact, we designed a Heuristic Allocation Initialization (HAI) algorithm to transfer computational resources between an under-allocated jobs and the shortest job.

We implemented a prototype, ReB, which aims to balance multiple-job resource allocations and minimize the average JCT. Extensive experiments were conducted in a Spark cluster, which consisted of 30 servers in six racks. Each server was equipped with an Intel Xeon E5-2650 2.2 GHz 12-core processor. Compared with the max-min fairness and shortest-job-optimal methods, ReB achieved a JCT reduction of 10%–30%. The main contributions of this paper are as follows.

- We observed that the underlying system overhead caused by the over-allocation of computational resources lengthens the JCT.
- We proposed a prediction model via machine learning, and found the optimal allocation without incurring an over-allocation problem.
- We designed an HAI algorithm to initialize resource allocation of multiple Spark jobs at startup.

The rest of this paper is organized as follows. Section 2 introduces related work about resource allocation and performance prediction. In Section 3, we rethink the over-allocation problem and present a prediction model with feature pairs. In Section 4, we propose a multiple-job allocation problem and design a heuristic algorithm for resource allocation. Section 5 shows the architecture and implementation details of ReB. Section 6 shows the evaluation results of extensive experiments. Sections 7 and 8 discuss and conclude this paper, respectively.

2 Related Work

2.1 Resource allocation for Spark jobs

Resource allocation is fundamental in distributed systems. For a Spark job, JCT is associated with the amount of allocated resource. Resource managers, such as Mesos and YARN, allocate computational resources to Spark jobs, and the amount of resource is customized by the users[7,8]. Classical resource allocation algorithms, such as the first-in-first-out method and shortest-job-first method, assume that resource demands and length of jobs are known in advance[9]. Nevertheless, these schedulers do not consider fairness and cause suboptimal JCT of individual jobs. To address this problem, the max-min fairness method and dominant resource fairness method were proposed[10,11].

Recent resource allocation methods have considered more details about Spark jobs. To enhance data locality, machine assignment is also one important problem involving resource allocation. Delay scheduling methods meet the requirements of data locality by optimizing task placement[12]. Tetris effectively addresses multiple resource packing problems by decreasing resource fragments and deadlocks[13]. Graphene aims to enhance resource utilization by filling in the virtual resource space with troublesome tasks first, then placing other tasks. The dependency and heterogeneous resource demands make it challenging to schedule tasks with high resource utilization.

2.2 Prediction for optimal configuration

Ernest ran the entire job on small datasets and tried to capture how the JCT of a job changed with the increasing size of the input dataset[14]. However, the model of Ernest was tightly bound to the application type, which limited the cross-application use of Ernest models. Bei et al.[15] proposed to construct two ensembles of performance models using a Random-Forest (RF) approach for the map and reduce stage, respectively. However, it is difficult to model the duration of each operation. Yu et al.[16] proposed a Hierarchical Modeling (HM), aiming to combine a number of individual sub-models in a hierarchical manner. However, the HM method was agnostic to the execution of various operations. The HM and the RF methods could not overcome the over-allocation problem in Spark jobs.

3 Prediction of a Single Spark Job

In this section, we designed a prediction model to determine the optimal allocation of a single Spark job before the job starts to execute. In practice, we did not directly predict the optimal allocation. Instead, the prediction model took two given allocations as the input and predicted the difference in their own JCTs. The JCT under the better allocation was smaller. According to the JCT difference, the better allocation was picked.

3.1 Design of DNN

Recently, deep learning techniques have been proven to outperform conventional machine learning models in many applications[17–19]. We chose a DNN as our predictor due to its capability of modeling complicated functions. The DNN consisted of multiple hidden layers, where each layer consisted of multiple neurons that were fully connected to those in the next layer. Although each neuron calculated the output through a simple activation function, all neurons were connected into a complex adaptive system, which obtained a powerful expressivity to model any function with unknown mathematical forms. Moreover, the DNN exhibited a better generalization ability, and achieved robust performance under noise data[20]. Deep learning is promising in overcoming the weakness of Ernest, RF, and HM methods.

The Bayesian regularization backpropagation algorithm was used as the training algorithm. It aimed to calculate the values of network parameters via the Levenberg-Marquardt optimization[21]. We set the learning rate to 0.05. The training process early-terminated when the number of training epochs reached 1000, or training loss decreased to 0.001.

3.2 Input features

To train the DNN, we collected completed Spark jobs as training data, called samples. We selected job configurations as input features of training data. The predicted value, i.e., the label of training data, was the JCT. The fundamental four input features included the size of the input data, the number of partitions, the number of cores, and the amount of memory. Additionally, the other three features involved the Directed Acyclic Graph (DAG) structure, which was acquired from the Spark job. DAGs indicated the workflow, along which the input was processed by various operations. Data were processed in parallel by each operation. The complexity of one job was associated with its DAG. As reported, production jobs at the 50th-percentile in Microsoft and BigBench exhibited the length from two to seven, the width from two to four, and the number of paths from two to six[22].

The last features were Spark operations. Input data of Spark jobs were processed by these operations one-by-one. For a Spark job, the type and the number of operations determined computation time, communication, and disk I/O overheads. For example, communication-intensive applications were inclined to frequently use network operations like Reduce. Since the number and the type of operations significantly impacted JCTs, we exploited operations to train a prediction model, which was aware of the underlying overhead.

3.3 Generating new samples with feature pairs

To train the prediction model, the value of features and the JCT of the completed Spark jobs were collected. However, a collected Spark job, called an “original sample”, was not directly used as a training sample. We combined two Spark jobs to generate a new training sample as follows: Given two Spark jobs, we took a pair of features as the input of the prediction model to predict the difference in JCTs of the two jobs.

Concretely, we combined two original samples, denoted by d_1 and d_2, to generate a new sample d'. Let $C(d_1)$ and $J(d_1)$ denote the job configuration and JCT of the sample d_1, respectively. We set the input features of the new sample d' to the combination of $C(d_1)$ and $C(d_2)$. The label of d' was set to $J(d_1) - J(d_2)$. That is, we input two different job configurations, $C(d_1)$ and $C(d_2)$, to predict the JCT difference. According to the JCT difference, we selected out the feature that achieved shorter JCT.

When we used the prediction model, we also generated test samples with feature pairs. Given a Spark job, we generated a group of test samples by changing only the number of allocated CPU cores (i.e., the values of other features were identical). We predicted the JCT difference to select the optimal number of CPU cores.

The advantages of feature pairs were as follows: First, the number of new training samples with feature pairs rapidly increased because of the combination of original samples. The number of new training samples was $\binom{n}{2}$, where n was the number of original samples. More samples were beneficial to achieve higher prediction accuracy. Second, we determined the optimal feature without directly predicting the JCTs. The prediction model with feature pairs combined the prediction and ranking together. This avoided the inaccuracy of ranking predicted JCTs and enhanced the accuracy in determining the optimal allocation. Moreover, the predicted JCT difference was used as the input of our heuristic algorithm, aiming to allocate resources of multiple Spark jobs.

4 Multi-Job Optimal Resource Allocation

In this section, we observe a multiple-job-optimal resource allocation problem from an illustrative example, propose a "naïve" method, called the shortest-job-optimal method, and further design a heuristic resource allocation method with a convergence proof.

4.1 Rethinking the multiple-job resource allocation problem

The profiles of two iterative jobs are given in Fig. 2a, which shows the number of iteration rounds, the number of CPU cores, and the job submission time. Job 1 and Job 2 differed only in the number of iteration rounds and submission times. For the two jobs, we illustrate three different schedules of the shortest-job-first, max-min fairness, and shortest-job-optimal methods in Figs. 2c–2e, respectively.

Note that the shortest-job-first and max-min fairness methods did not know the optimal allocation a priori. Thus, the two methods allocated CPU cores according to the requested cores of Job 1 and Job 2. The shortest-job-optimal method was under the support of our prediction method and determined the optimal allocation through information in Fig. 2b. The two jobs requested 80 cores, but the optimal setting of CPU cores was 60. Figures 2c and 2d show that the shortest-job-first and max-min fairness methods allocated 80 cores to Job 1 before the 8th second. After Job 2 emerged, the shortest-job-first method preferred to meet the request of Job 2 and allocated the remaining 20 cores to Job 1. In this case, Job 2 was over-allocated but Job 1 was under-allocated. The average JCT was 30 s. In Fig. 2d, the max-min fairness method allocated the same number of cores to two jobs after Job 2 was submitted. The average JCT was 24 s. In summary, both the shortest-job-first and max-min fairness methods were suboptimal because of the under-/over-allocation.

4.2 "Naïve" shortest-job-optimal method

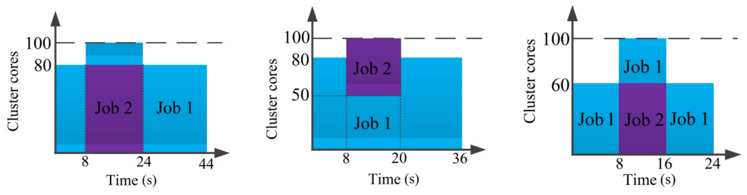
As a variant of the shortest-job-first method, the shortest-job-optimal method aims to optimize the allocation of the shortest job under the support of our prediction model. Algorithm 1 showed the shortest-job-optimal method. Nevertheless, we observed the shortest-job-optimal method to be suboptimal in

	Job 1	Job 2
Submission time (s)	0	8
Iteration rounds	10	4
Requested cores	80	80

(a) Profiles of Job 1 and Job 2.

Allocated cores	Duration/Iteration	
	Job 1 (s)	Job 2 (s)
80 (requested)	4	4
70	3	3
60 (optimal)	2	2
50	3	3
40	4	4
20	5	5

(b) Duration per iteration varies with CPU cores.



(c) Shortest-job-first method. The average JCT is 30 s. (d) Max-min fairness method. The average JCT is 24 s. (e) Shortest-job-optimal method. The average JCT is 16 s.

Fig. 2 Illustrated examples of scheduling Job 1 and Job 2 by the shortest-job-first, max-min fairness, and shortest-job-optimal methods, respectively.

Algorithm 1 Naïve shortest-job-optimal method

```
Input: The available resource capacity $C$, a batch of jobs $J = \{J_1, J_2, \ldots, J_n\}$, their optimal allocations $A^{\rm OPT} = \{A_1^{\rm OPT}, A_2^{\rm OPT}, \ldots, A_n^{\rm OPT}\}$, and the corresponding JCTs $\{D_1, D_2, \ldots, D_n\}$.
1: Sort all jobs $J$ in increasing order of their JCTs
2: for the $i$-th job in $J$ do
3:     if remaining resource $C$ is sufficient then
4:         Allocate $A_i^{\rm OPT}$ to $J_i$
5:         $C \leftarrow C - A_i^{\rm OPT}$
6:     else
7:         Allocate $C$ to $J_i$
```

multiple-job cases.

We executed four PageRank jobs concurrently in a small-scale benchmark, where 120 cores were available in total. The four jobs were completely identical to the PageRank job in Fig. 3a and were submitted at the same time. Figure 3a shows the changing JCT of a PageRank job when the number of CPU cores increased. The optimal number of cores was 40. We leveraged the shortest-job-optimal method. Figure 3 shows the number of cores that were allocated to four PageRank jobs. Figure 3b shows details of the shortest-job-optimal method. Note that not all jobs were optimal-allocated because CPU cores were not sufficient. Job 1, Job 2, and Job 3 were all allocated at the beginning. However, Job 4 was delayed owing to insufficient resource and did not start until Job 1 completed and released the resource.

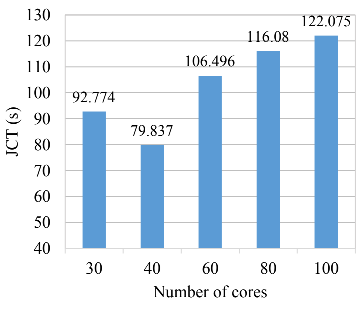
We compared the shortest-job-optimal method with a heuristic method. Figure 3c shows that the heuristic method achieved fewer JCTs. The heuristic method preferred to under-allocate 30 cores to three PageRank jobs. The above evaluation revealed that the average JCT of the shortest-job-optimal method may have been prolonged, although the allocation of partial jobs was optimal.

4.3 Heuristic method

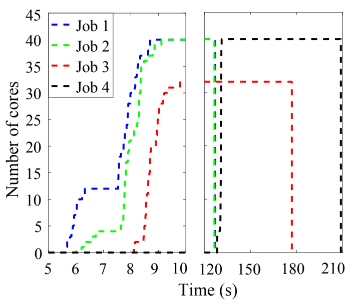
A drawback of the shortest-job-optimal method is that only a few jobs are optimally allocated, but the majority of jobs suffer from serious under-allocation. The fundamental problem was to calculate a schedule of balancing resource allocations among multiple jobs, aiming to minimize the average JCT. The schedule was formulated as a mixed discrete and non-convex problem, which was NP-hard[23]. Fortunately, a near-optimal schedule was computable under the support of the prediction model.

We designed an HAI method, which fine-tuned the amount of allocated resource by continuously transferring a few CPU cores from a job, called a donator, to another job, called an acceptor. The transfer changed the value of the average JCT. We stopped transferring CPU cores until the average JCT did not decrease. This involved two challenges, including (1) which jobs were transferred from/to, and (2) how many cores, in total, should have been transferred.

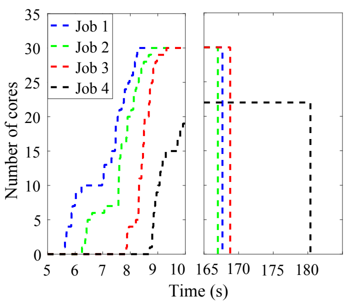
Algorithm 2 showed the main steps of the HAI algorithm. The input was a parameter θ, a batch of jobs J, and the predicted change in the values of JCTs, denoted by $\{\Delta{\rm JCT}_1, \Delta{\rm JCT}_2, \ldots, \Delta{\rm JCT}_n\}$. The output of Algorithm 2 was the allocation of each job, denoted by $A = \{A_1, A_2, \ldots, A_n\}$. In the beginning, we leveraged the shortest-job-optimal method to calculate the allocation of all jobs and classify all jobs into two groups, J_1 and J_2. For each optimally-allocated job u in J_1, we subtracted θ cores from its current allocation A_u and predicted the increment of its JCT, denoted by $\Delta{\rm JCT}_u(A_u, A_u - \theta)$. The job, which exhibited the minimum increment of its JCT, was selected as the donator. That is, $\Delta{\rm JCT}_{\rm donator} =$



(a) JCT of a PageRank job changes with increasing CPU cores. The setting of 40 cores is the optimal. The setting of 30 cores causes the under-allocation.



(b) Shortest-job-optimal method prefers the optimal allocation of 40 cores. Job 1, Job 2, and Job 3 are allocated 40, 40, and 32 cores, respectively. However, Job 4 is delayed.



(c) Heuristic method does not delay any job for the optimal allocation of individual jobs. Job 1, Job 2, and Job 3 are allocated 30 cores. Job 4 are allocated 22 cores.

Fig. 3 Micro benchmark shows that the shortest-job-optimal method was suboptimal. The same 4 jobs (Job 1, Job 2, Job 3, and Job 4) were a PageRank job in (a) and submitted at the same time. The optimal allocation was 40 cores.

$\min_{i \in J_1} \Delta\text{JCT}_i(A_i, A_i - \theta)$. For each non-optimally-allocated job v in J_2, we added θ cores to its current allocation A_v and predicted the decrease in its JCT, denoted by $\Delta\text{JCT}_v(A_v, A_v + \theta)$. The job, which exhibited the maximum decrement of its JCT, was selected as the acceptor. That is, $\Delta\text{JCT}_{\text{acceptor}} = \max_{i \in J_2} \Delta\text{JCT}_i(A_i, A_i + \theta)$. If the decrement of the acceptor's JCT was larger than the increment of the donator's JCT, i.e., $\Delta\text{JCT}_{\text{acceptor}} > \Delta\text{JCT}_{\text{donator}}$, it could be guaranteed that the sum of all JCTs decreased after transferring θ cores from the donator to the acceptor. The transfer was executed continuously until the transfer between any donators and any acceptors could not decrease the sum of all JCTs. Then, Algorithm 2 terminated.

4.4 Analysis of the time complexity

Considering the process of transferring cores, Algorithm 2 was an iterative algorithm. Next, we proved that Algorithm 2 was convergent when J_1 and J_2 contained only one job. We calculated the maximum number of transfers in the worst case, indicating the time complexity. Then, we expanded the proof for the multiple-job case.

Theorem 1 Given any two jobs, Job 1 and Job 2, the resource allocation of Job 1 is the optimal, denoted by A_1. Job 2 is under-allocated, denoted by A_2. By repeatedly transferring θ cores from Job 1 to Job 2, Algorithm 2 is convergent. In the worst case, the maximum round of iterations is $\lfloor A_1/\theta \rfloor$.

Algorithm 2 Heuristic allocation initialization

```
Input: A parameter $\theta$, a batch of jobs $J$, and their predicted
       JCT differences $\{\Delta\text{JCT}_1, \Delta\text{JCT}_2, \ldots, \Delta\text{JCT}_n\}$.
Output: Allocations $A = \{A_1, A_2, \ldots, A_n\}$.
 1: toBeContinued = ture, $J_1 = \varnothing$, $J_2 = \varnothing$
 2: Initialize $A$ by the shortest-job-optimal method
 3: for Each job, $i = 1$ to $n$ do
 4:     if Job $i$ is optimal-allocated then
 5:         Classify Job $i$ into $J_1$
 6:     else
 7:         Classify Job $i$ into $J_2$
 8: while toBeContinued = true do
 9:     for Each job $u$ in $J_1$ do
10:         $D_u = \Delta\text{JCT}_u(A_u, A_u - \theta)$
11:     The donator is the job with $\min\{D_1, \ldots, D_u, \ldots\}$
12:     for Each job $v$ in $J_2$ do after adding $\theta$ cores
13:         $D_v = \Delta\text{JCT}_v(A_v, A_v + \theta)$
14:     The acceptor is the job with $\max\{D_1, \ldots, D_v, \ldots\}$
15:     if $\min_{u \in J_1} D_u < \max_{v \in J_2} D_v$ then
16:         $A_{\text{donator}} = A_{\text{donator}} - \theta$
17:         $A_{\text{acceptor}} = A_{\text{acceptor}} + \theta$
18:     else
19:         toBeContinued = false
```

Proof: Algorithm 2 stopped transferring as long as $\min\{D_u | u \in J_1\} < \max\{D_v | v \in J_2\}$ was satisfied. Here, u and v were referred to as Job 1 and Job 2, respectively. Thus, we needed to prove $D_1 < D_2$. Let $\mathcal{J}_i$ denote the sum of all JCTs at the i-th transfer. For the first transfer, the sum of JCT was $\mathcal{J}_1 = \text{JCT}_1(A_1 - \theta) + \text{JCT}_2(A_2 + \theta)$. Similarly, for the k-th transfer, the sum of JCTs was $\mathcal{J}_k = \text{JCT}_1(A_1 - k \times \theta) + \text{JCT}_2(A_2 + k \times \theta)$. Note that, the allocation of Job 1 was always positive and $A_1 - k \times \theta > 0$. Thus, the maximum number of transferring cores was $\lfloor A_1/\theta \rfloor$. When Algorithm 2 stopped at the k-th transfer, we inferred $\mathcal{J}_k < \mathcal{J}_{k+1}$. Considering $\mathcal{J}_{k+1} = \text{JCT}_1(A_1 - (k+1) \times \theta) + \text{JCT}_2(A_2 + (k+1) \times \theta)$, we had

$$\text{JCT}_1(A_1 - k \times \theta) + \text{JCT}_2(A_2 + k \times \theta) <$$
$$\text{JCT}_1(A_1 - (k+1) \times \theta) + \text{JCT}_2(A_2 + (k+1) \times \theta).$$

We transformed the above inequality as follows,

$$\text{JCT}_1(A_1 - k \times \theta) - \text{JCT}_1(A_1 - (k+1) \times \theta) <$$
$$\text{JCT}_2(A_2 + (k+1) \times \theta) - \text{JCT}_2(A_2 + k \times \theta).$$

We knew that $D_1 = \Delta\text{JCT}_1(A_1 - k \times \theta, A_1 - (k+1) \times \theta) = \text{JCT}_1(A_1 - k \times \theta) - \text{JCT}_1(A_1 - (k+1) \times \theta)$ and similarly, $D_2 = \Delta\text{JCT}_2(A_2 + (k+1) \times \theta, A_2 + k \times \theta) = \text{JCT}_2(A_2 + (k+1) \times \theta) - \text{JCT}_2(A_2 + k \times \theta)$. Thus, $D_1 < D_2$.

As mentioned, the maximum number of transferring cores was $\lfloor A_1/\theta \rfloor$ when almost all cores of Job 1 were transferred to Job 2. Theorem 1 is proved. ■

The Proof assumed that sets J_1 and J_2 contained one job, respectively. When the number of jobs in J_1 or J_2 was over 2, the convergence proof was similar. We iteratively transferred cores from jobs in J_1 to those in J_2 until the sum of JCTs did not decrease. The transfers stopped until the sum of JCTs did not decrease. Let A_i^{optimal} denote the optimal allocation of the i-th job in J_1. When all jobs in J_1 were transferred, the number of transfers was at a maximum, i.e., $\sum_{i \in J_1} \lfloor A_i^{\text{optimal}}/\theta \rfloor$. Moreover, the sum of the CPU cores, which were allocated to jobs in J_1, was the capacity of CPU cores, denoted by C. In multiple-job cases, we had that the maximum number of transferring times was $\sum_{i \in J_1} \lfloor A_i^{\text{optimal}}/\theta \rfloor \approx \lfloor C/\theta \rfloor$. The value was also the times of executing the "while" loop in the worst case. On the other hand, the time complexity of each execution of the "while" loop was $O(n)$, where n was the number of jobs. Therefore, the time complexity of the "while" loop was $O(n \times \lfloor C/\theta \rfloor)$. The time complexity of the shortest-job-optimal method was $O(n)$. The complexity of Algorithm 2 was the sum of $O(n \times \lfloor C/\theta \rfloor)$ and $O(n)$.

5 ReB Architecture

We implemented a prototype of ReB by extending a few modules of Apache Spark. ReB included two main modules, a prediction and and an allocation module. The prediction module involved a data source and a learning algorithm. The allocation module aimed to balance the resource allocation of different jobs under the support of the prediction module.

Figure 4 shows the workflow of ReB. Before jobs were submitted, we collected training data from Spark jobs. The input features and output label of training data were job configurations and JCTs, respectively. Then, a DNN-based model was trained with the collected data. The training process was done offline. The prediction model was used when Spark jobs were submitted

ReB analyzes Spark jobs and collects job configurations, such as the size of input data, DAG structures, etc. The configuration information was input into the prediction model. The prediction model predicted the varying JCT of each submitted job when the number of CPU cores changed. Then, the JCT was used as the input of Algorithm 2. The initial allocation of submitted jobs was calculated. These jobs were then submitted to the Spark master and started to execute.

6 Performance Evaluation

In this section, we evaluate the prediction accuracy of our prediction model, the performance improvement in terms of the JCT, and the algorithm overhead of Algorithm 2.

6.1 Experiment evaluation of multiple-job scenarios

We conducted extensive experiments to evaluate the performance of ReB, including the average JCT, cumulative number of jobs, and the varying scale of computational resources. Cluster experiments were conducted in a Spark cluster, which consisted of 30 servers in six racks. Each server was equipped with an Intel Xeon E5-2650 2.2 GHz 12-core processor.

6.1.1 Settings and metrics

For multiple-job cases, we compared ReB with the shortest-job-optimal and max-min fairness methods. Both the shortest-job-optimal method and ReB worked under the support of the prediction model. For ReB, we set θ to 2 cores. The shortest-job-optimal method optimized the allocation of the shortest job. The max-min fairness method allocated resource in a fair manner so that each job could not starve. We used PageRank as the basic job and submitted multiple PageRank jobs at the same time. We employed the factor of improvement to evaluate the average JCT, as well as the makespan when the number of jobs increased from 10 to 50. As the legend indicates, the factor of ReB over Max-min fairness denoted the improvement percentage of ReB to the max-min fairness method.

6.1.2 Evaluation of the average JCT

Figure 5 shows the average JCT improvement factor and absolute JCTs using the bars and dotted lines, respectively. The max-min fairness method performed

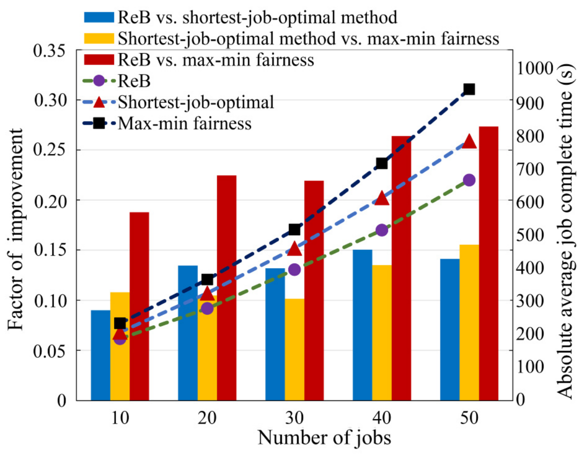


Fig. 5 Average JCT of ReB versus the shortest-job-optimal and max-min fairness methods.

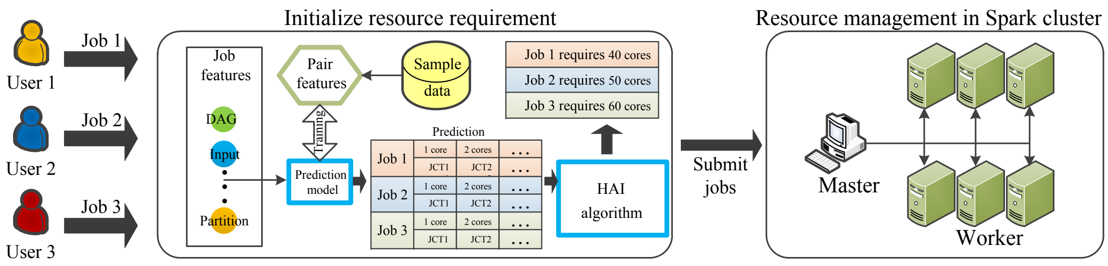


Fig. 4 Workflow of the ReB.

the worst as each job suffered severe under-allocation owing to fairness. ReB and the shortest-job-optimal methods exhibited shorter average JCTs. When the number of jobs was 10, the shortest-job-optimal method performed close to ReB. Nevertheless, the improvement factor of ReB over Shortest-job-optimal method increased with the number of jobs. Results showed that ReB outperformed the shortest-job-optimal method when more jobs contended for computational resources. We inferred that the number of delayed jobs also increased when more jobs emerged as the optimal allocation of individual jobs occupied considerable resource. That is, the disadvantage of the shortest-job-optimal method became more evident. In summary, ReB achieved around 10%–25% improvement.

6.1.3 Evaluation of the JCT distribution

We utilized a boxplot to show details of JCTs when 50 jobs were submitted in a batch. In Fig. 6, each boxplot reflects the smallest, largest, average, and median JCT. As the boxes show, most of the JCTs were in the range of first quartile to third quartile. It was interesting to note that the boxes of the shortest-job-optimal and max-min fairness methods were the longest and shortest, respectively. We noted that the large boxes indicated a sparse JCT distribution, which potentially increased the makespan. To the contrary, the dense JCT distribution was a short box, and the average JCT was also large. For ReB, most jobs completed from 400 s to 1000 s, which was earlier than for the other methods. ReB achieved the least average JCT.

In summary, the max-min fairness method achieved the shortest makespan but the longest average JCT. ReB achieved the best average JCT and its makespan was close to that of the max-min fairness method. We attributed the benefits of ReB to three reasons. First, ReB avoided over-/under-allocation for optimization of the single-job allocation. Second, ReB preferred the globally-optimal allocation rather than that of the individual jobs. Third, ReB decreased the number of jobs that were delayed by the resource contention. When the number of jobs increased, the performance improvement of ReB was more evident.

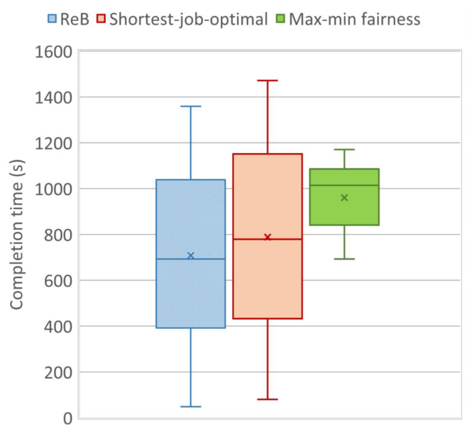


Fig. 6 Completion times of 50 jobs scheduled by ReB, the shortest-job-optimal, and max-min fairness methods.

6.1.4 Evaluation of the scale of computation resource

The above experiments were conducted using 200 cores and 500 GB RAM. In order to evaluate the sensitivity of ReB against the change in the cluster scale, we increased the number of total cores from 200 to 400. Figure 7 shows that the number of cores imposed a significant impact on the improvement factor. It was clear that the improvement factor of ReB over the shortest-job-optimal method decreased with increasing number of cores. Although the average JCT under the two methods decreased, the shortest-job-optimal method performed better when computational resources increased. We inferred that the increase in computational resources weakened the resource contention among the multiple jobs, and thus reduced the performance improvement of ReB. The improvement factor of ReB over the max-min fairness method also exhibited a similar trend. The scalability showed that ReB performed better when computational resources were insufficient, i.e., the resource contention between multiple jobs was very strong.

6.1.5 Evaluation of the sensitivity of input size on jobs

In addition to the cluster scale, we also changed the input size of each job from 10 GB to 50 GB. Figure 8 shows that the improvement factor of average JCT. We can see that the shortest-job-optimal method achieved a shorter JCT than the max-min fairness method.

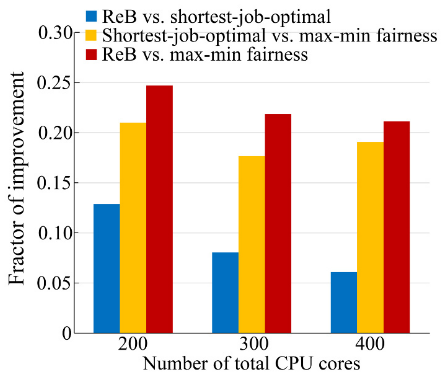


Fig. 7 Comparison of the improvement in average JCT when the number of CPU cores increased from 200 to 400.

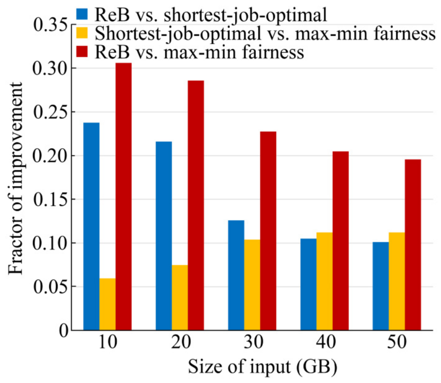


Fig. 8 Comparison of the improvement of average JCT when the input size increased from 10 GB to 50 GB.

The improvement factor of the shortest-job-optimal method vs. the max-min fairness method decreased by around 25% to 10%. This indicated that the shortest-job-optimal method was more disadvantageous in scheduling large jobs. We inferred that the larger amount of input data caused longer starvation as more jobs suffered from serious under-allocation. Nevertheless, ReB achieved the greatest improvement versus the other two methods. We conclude that ReB is robust for changing input size.

6.2 Evaluation of the algorithm overhead

We further measured the time complexity of Algorithm 2 under 200 cores and different values of θ when the number of jobs ranged from 10 to 50. Figure 9 shows the time complexity when the value of θ is set to 1, 5, and 10, respectively. The algorithm overhead was approximately one second when the number of jobs was 10. Nevertheless, the overhead rapidly increased when the number of jobs increased. We conclude that a larger θ incurs less algorithm overhead. The difference in algorithm overheads using different θ under 50 jobs became huge. Although the algorithm overhead with a small θ is high, its performance was close to that of the optimal solution. In future work, we will consider the setting of adaptive θ across iterations in order to accelerate the execution of Algorithm 2.

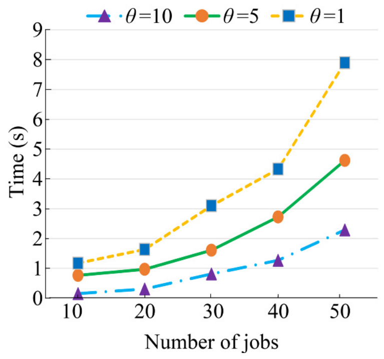


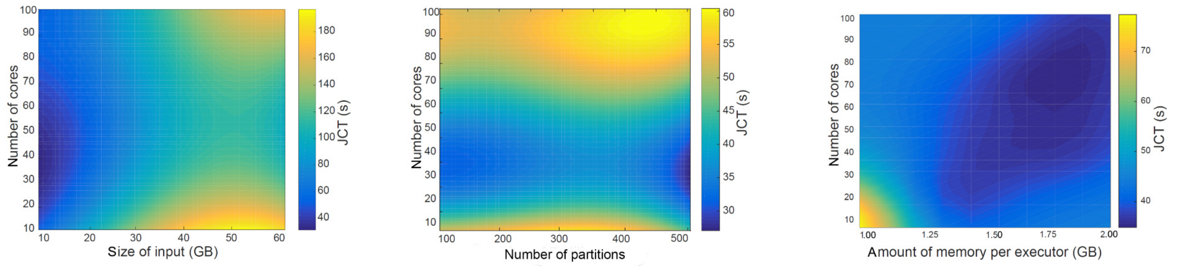
Fig. 9 Overhead of Algorithm 2 when jobs increased from 10 to 50.

7 Discussion

As mentioned, we had four fundamental features, i.e., the number of CPU cores, the size of input data, the number of partitions, and the amount of memory per executor. We leveraged the prediction model to predict changing JCT when two of the features changed. Furthermore, we plotted the changing trend in predicted JCTs.

Figure 10a shows that the JCT changed with the number of cores and input size. Different colors denote the length of JCTs. The deep blue region indicates that 40 cores were optimal for the workload of 10 GB. When the input size increased, the JCT also increased. When the input size was 60 GB, the optimal allocation was around approximately 50 cores.

Figure 10b shows that the JCT changed with the number of cores and the partition. When the number of partitions increased from 100 to 500, the optimal allocation remained the same. This indicated that the number of partitions did not directly impact the optimal allocation. Figure 10c shows the impact of the memory and CPU cores. Larger memory was more advantageous.



(a) Number of cores increases from 10 to 100, the input size increases from 10 GB to 60 GB.

(b) Number of cores increases from 10 to 100, the number of partitions increase from 100 to 500.

(c) Number of cores increases from 10 to 100, the amount of memory per executor increases from 1 to 2 GB.

Fig. 10 Impact of partitions, cores, input size, and memory on the JCT prediction.

8 Conclusion

ReB is a smart scheduler for balancing resource allocation for Spark jobs. It trains the prediction model for the optimal allocation by using operation features and reforming samples pairwise. ReB balances the global resource allocation involving multiple Spark jobs. Our experiments demonstrated that ReB outperforms the shortest-job-optimal and max-min fairness methods. ReB decreases the average JCT by 15%–30%.

Acknowledgment

This work was supported in part by the National Key R&D Program of China (No. 2018YFB2101100), the National Natural Science Foundation of China (Nos. 61932001 and 61872376), and Hunan Provincial Innovation Foundation For Postgraduate.

References

[1] C. Mosharaf, Z. Matei, M. Justin, J. Michael, and S. Ion, Managing data transfers in computer clusters with Orchestra, *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[2] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, Scale and performance in a distributed file system, *ACM Transaction on Computer System*, vol. 6, no. 1, pp. 51–81, 1988.

[3] D. P. Woodruff and Q. Zhang, When distributed computation is communication expensive, *Distributed Computing*, vol. 30, no. 5, pp. 309–323, 2017.

[4] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, in *Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, CA, USA, 2004, pp. 137–150.

[5] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, Cloudscale: Elastic resource scaling for multi-tenant cloud systems, in *Proceedings of ACM Symposium on Cloud Computing (SOCC'11)*, Cascais, Portugal, 2011, pp. 5–17.

[6] C. Delimitrou and C. Kozyrakis, Quasar: Resource-efficient and QoS-aware cluster management, in *Proceedings of ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, Salt Lake City, UT, USA, 2014, pp. 127–144.

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, Mesos: A platform for fine-grained resource sharing in the data center, in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, USA, 2011, pp. 429–483.

[8] 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 *Proceedings of ACM Symposium on Cloud Computing (SOCC'13)*, Santa Clara, CA, USA, 2013, pp. 1–16.

[9] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, Resource management with deep reinforcement learning, in *Proceedings of ACM HotNet Workshop on Hot Topics in Networks (HotNet'16)*, Atlanta, GA, USA, 2016, pp. 50–56.

[10] T. Bonald, L. Massouli, A. Prouti, and J. T. Virtamo, A queueing analysis of max-min fairness, proportional fairness and balanced fairness, *Queueing Systems*, vol. 53, nos. 1&2, pp. 65–84, 2006.

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, Dominant resource fairness: Fair allocation of multiple resource types, in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Boston, MA, USA, 2013, pp. 323–336.

[12] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling, in *Proceedings of European Conference on Computer Systems (EuroSys'10)*, Paris, France, 2010, pp. 265–278.

[13] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, Multi-resource packing for cluster schedulers, in *Proceedings of ACM Special Interest Group on Data Communication (SIGCOMM'14)*, Chicago, IL, USA, 2014, pp. 455–466.

[14] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, Ernest: Efficient performance prediction for large-scale advanced analytics, in *Proceedings of USENIX Symposium on Networked Systems Design and Implementatio (NSDI'16)*, Santa Clara, CA, USA, 2016, pp. 363–378.

[15] Z. Bei, Z. Yu, H. Zhang, W. Xiong, C. Xu, L. Eeckhout, and S. Feng, RFHOC: A random-forest approach to auto-tuning Hadoop's configuration, *IEEE Transaction on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1470–1483, 2016.

[16] Z. Yu, Z. Bei, and X. Qian, Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing, in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, Williamsburg, VA, USA, 2018, pp. 564–577.

[17] C. Re, D. Agrawal, M. Balazinska, M. J. Cafarella, M. I. Jordan, T. Kraska, and R. Ramakrishnan, Machine learning and databases: The sound of things to come or a cacophony of hype? in *Proceedings of ACM International Conference on Management of Data (SIGMOD'15)*, Melbourne, Australia, 2015, pp. 283–284.

[18] L. Sun, S. Sun, T. Wang, J. Li, and J. Lin, Parallel ADR detection based on spark and BCPNN, *Tsinghua Science and Technology*, vol. 24, no. 2, pp. 195–206, 2019.

[19] X. Ye, X. Chen, D. Liu, W. Wang, L. Yang, G. Liang, and G. Shao, Notice of retraction: Efficient feature

extraction using Apache Spark for network behavior anomaly detection, *Tsinghua Science and Technology*, vol. 23, no. 5, pp. 561–573, 2018.

[20] M. Wang, Y. Cui, S. Xiao, X. Wang, D. Yang, K. Chen, and J. Zhu, Neural network meets DCN: Traffic-driven topology adaptation with deep learning, in *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'18)*, Irvine, CA, USA, 2018, pp. 97–99.

[21] N. Yamashita and M. Fukushima, On the rate of convergence of the Levenberg-Marquardt method, *Springer Computing*, vol. 15, pp. 239–249, 2001.

[22] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, Altruistic scheduling in multi-resource clusters, in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, USA, 2016, pp. 65–80.

[23] D. Fooladivanda, A. A. Daoud, and C. Rosenberg, Joint channel allocation and user association for heterogeneous wireless cellular networks, *IEEE Transaction on Wireless Communications*, vol. 12, no. 1, pp. 248–257, 2011.

Zhiyao Hu received the BS degree from Beijing Institute of Technology. He is now a PhD candidate at the School of Computer, National University of Defense Technology, Changsha, China. His main research interests include distributed computing, data-parallel distributed framework, and machine learning.

Dongsheng Li received the BS and PhD degrees from National University of Defense Technology, Changsha, China, in 1999 and 2005, respectively. He is now a professor at the National Lab for Parallel and Distributed Processing. His research interests include distributed computing, cloud computing, and big Data.

Deke Guo received the BS and PhD degrees from National University of Defense Technology, Changsha, China, in 2001 and 2008, respectively. He is now a professor at the School of System Engineering, National University of Defense Technology. His research interests include computer networks, edge computing, and data center networking.