J. Parallel Distrib. Comput. 90–91 (2016) 52–66

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Joint scheduling of MapReduce jobs with servers: Performance bounds and experiments

Xiao Ling [a], Yi Yuan [b], Dan Wang [b], Jiangchuan Liu [c], Jiahai Yang [a,*]

[a] Tsinghua National Laboratory for Information Science and Technology, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China
[b] Department of Computing Science, The Hong Kong Polytechnic University, Hong Kong
[c] Department of Computing Science, Simon Fraser University, British Columbia, Canada

HIGHLIGHTS

- We investigate a schedule problem for MapReduce-like frameworks by taking server assignment into consideration.
- We formulate the MapReduce server-job organizer problem (MSJO) and show that it is NP-complete.
- We propose a 3-approximation algorithm and a fast heuristic design to address the MSJO problem.
- We implement our algorithms and some state-of-the-art algorithms on Amazon EC2 with deploying schedulers in Hadoop.
- By comprehensive simulations and experiments, the results show that our algorithm outperforms other classical strategies.

ARTICLE INFO

Article history:
Received 29 August 2015
Received in revised form
31 December 2015
Accepted 21 February 2016
Available online 2 March 2016

Keywords:
MapReduce
Scheduling
Server assignment
NP-complete
Fast heuristic

ABSTRACT

MapReduce-like frameworks have achieved tremendous success for large-scale data processing in data centers. A key feature distinguishing MapReduce from previous parallel models is that it interleaves parallel and sequential computation. Past schemes, and especially their theoretical bounds, on general parallel models are therefore, unlikely to be applied to MapReduce directly. There are many recent studies on MapReduce job and task scheduling. These studies assume that the servers are assigned in advance. In current data centers, multiple MapReduce jobs of different importance levels run together. In this paper, we investigate a schedule problem for MapReduce taking server assignment into consideration as well. We formulate a MapReduce server-job organizer problem (MSJO) and show that it is NP-complete. We develop a 3-approximation algorithm and a fast heuristic design. Moreover, we further propose a novel fine-grained practical algorithm for general MapReduce-like task scheduling problem. Finally, we evaluate our algorithms through both simulations and experiments on Amazon EC2 with an implementation with Hadoop. The results confirm the superiority of our algorithms.

1. Introduction

Recently the amount of data of various applications has increased beyond the processing capability of single machines. To cope with such data, scale out parallel processing is widely accepted. MapReduce [11], the de facto standard framework in parallel processing for big data applications, has become widely adopted. Nevertheless, MapReduce framework is also criticized for its inefficiency in performance and as "a major step backward" [14]. This is partially because that, performance-wise, the MapReduce framework has not been deeply studied enough as compared to decades of study and fine-tune of other conventional systems. As a consequence, there are many recent studies in improving MapReduce performance.

MapReduce breaks down a job into map tasks and reduce tasks. These tasks are parallelized across server clusters.[1] Although map tasks and reduce tasks overlap partly in the real Hadoop scheduling mechanism, researchers [18,19] generally assume that reduce

* Corresponding author.
E-mail addresses: lxcernet@gmail.com (X. Ling), robertyi@163.com (Y. Yuan), csdwang@comp.polyu.edu.hk (D. Wang), jcliu@cs.sfu.ca (J. Liu), yang@cernet.edu.cn (J. Yang).

[1] The server clusters here are meant to be general; it can either be data center servers or cloud virtual machines.

http://dx.doi.org/10.1016/j.jpdc.2016.02.002

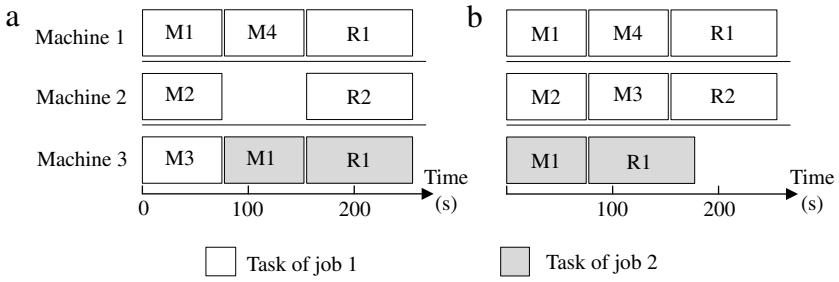

Fig. 1. Impact of server assignment. (a) Without server assignment, Hadoop default strategy. (b) Joint considering of server assignment.

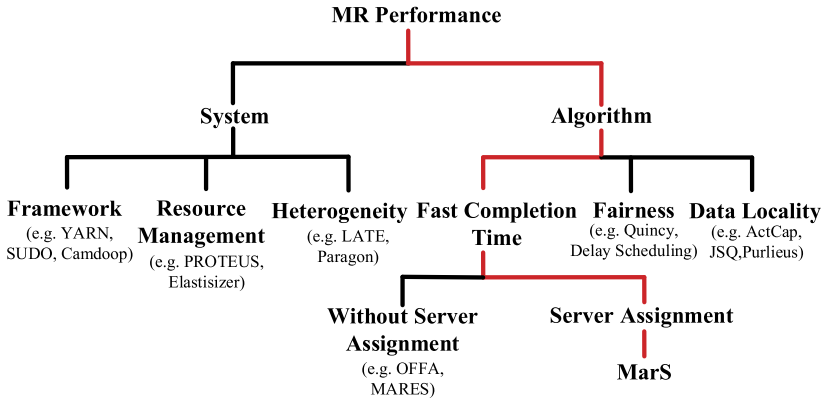

Fig. 2. Design space of MarS.

tasks will not start until the accomplishment of map tasks because reduce tasks rely on intermediate data produced by the map tasks. This is a parallel–sequential structure. In current practice, multiple MapReduce jobs are scheduled simultaneously to efficiently utilize the computation resources in the data center servers. It is a non-trivial task to find a good schedule for multiple MapReduce jobs and tasks running on different servers. There are a great number of studies on general parallel processing scheduling in the past decades. Nevertheless, whether these techniques can be applied directly in the MapReduce framework is not clear; and especially, their results on theoretical bounds are unlikely to be translated.

In this paper, we conduct research in this direction. There are recent studies on MapReduce scheduling [7,8]. As an example, an algorithm is developed in [8] for joint scheduling of processing and shuffle phases and it achieves an 8-approximation. To the best of our knowledge, previous studies commonly assume that the servers are assigned. That is, they assume that tasks in MapReduce jobs are first assigned to the servers, and their scheduling is conducted to manage the sequences of the map and reduce tasks in each job. It is not clear whether the scheduling on map and reduce tasks will be affected in a situation where the server assignment is "less good". We illustrate this impact by a toy example in Fig. 1. There are three machines and two jobs. Job 1 has 4 map tasks and 2 reduce tasks. Job 2 has 1 map task and 1 reduce task. Assume the processing time to be 75 s for every single map task and 100 s for every single reduce task. If server assignment is not considered, it will result in Fig. 1(a), which follows the default FIFO strategy of Hadoop [3]. However, if we jointly consider server assignment, we can achieve a schedule shown in Fig. 1(b). It is easy to see that the completion time of job 2 in Fig. 1(a) is 250 s and in Fig. 1(b) is 175 s, a 30% improvement.

In this paper, we fill in this blank by jointly consider server assignments and MapReduce jobs (and the associated tasks). To systematically study this problem, we formulate a unique MapReduce server-job organizer problem (MSJO). Note that the MSJO we discuss is the general case where the jobs can have different weights. We show that MSJO is NP-complete and we develop a 3-approximation algorithm. This approximation algorithm, though polynomial, has certain complexity in solving an LP-subroutine. Therefore, we further develop a fast heuristic. We evaluate our algorithm through extensive simulations. Our algorithm can outperforms the state-of-the-art algorithms by 40% in terms of total weighted job completion time. We further implement our algorithm in Hadoop and evaluate our algorithm using experiments in Amazon EC2 [1]. The experiment results confirm the advantage of our algorithm.

The rest of the paper is organized as follows. We discuss background in Section 2. We formulate the MSJO problem and analyze its complexity in Section 3. In Section 4, we present several algorithms. We evaluate our algorithms in Section 5. In Section 6, we show an implementation of our scheme in Hadoop and our evaluation in Amazon EC2. Finally, Section 7 summarizes the paper.

2. Related work

Due to the wide usage of MapReduce systems, there is a flourish of studies on understanding MapReduce performance and many developed various improvement schemes. One classification divides the view point by system and algorithm. Our work belongs to algorithm research and we categorize this in Fig. 2.

From system point of view: (1) Framework. There are many valuable advances on improving MapReduce framework. For example, Apache YARN [5] is a new kind of Hadoop resource manager, which can provide unified resource management and scheduling for the above big data applications. Mesos [4] is built using the same principles as the Linux kernel and provides applications (e.g., Hadoop, Spark, Kafka) with APIs for resource management and scheduling across entire datacenter and cloud environments. However, these open source programs focus on

the cloud resource management and system implementation, but not the optimal job/task scheduling issues. Zhang et al. [40] identified useful functional properties for user-defined functions and proposed an optimization framework SUDO that reasons about data-partition partition properties, functional properties, and data shuffling. Costa et al. [9] built a MapReduce-like system Camdoop to decrease the traffic by using a direct-connect network topology with servers. Zhang et al. [39] designed MIMP, a minimal interference, maximal progress scheduling system which manages both VM CPU scheduling and Hadoop job scheduling to reduce interference and increase overall efficiency. Li et al. [24] presented an efficient scheduling framework WOHA for deadline-aware MapReduce workflows. Literature [23] introduces the technique of packing server to convert independent task set schedulability bounds to MapReduce workflows schedulability bounds for real-time analytic applications. Our attention is focused on another aspect, which means we try to optimize the MapReduce performance bound in terms of the total weighted job completion time based on LP relaxation. (2) Resource Management. For example, Xie et al. [35] observed that the MapReduce applications have different network bandwidth requirements at different stages of the job execution, then proposed a model network bandwidth requirements of MapReduce jobs and a system PROTEUS to maximize number of accommodated jobs. Herodotou et al. [20] developed Elastisizer to which users can express their cluster sizing problems as queries in a declarative fashion, and can provide reliable answers to these queries using an automated technique and provide nonexpert users with a good combination of cluster resource and job configuration settings to meet their needs. In [13], Quasar, a cluster management system, is proposed to increase resource utilization while providing consistently high application performance. It uses fast classification techniques to determine the impact of different resource allocations and assignments on workload performance. (3) Heterogeneity. Zaharia et al. [38] designed a robust scheduling algorithm LATE to address the performance issues incurred by speculative execution in heterogeneous environment. In [12], Paragon, an online and QoS-aware scheduler proposed for heterogeneous clusters, includes a greedy server scheduler which minimizes interference and increases server utilization. Besides, outliers are considered in [2], which proposed Mantri to monitor tasks and cull outliers using cause- and resource-aware techniques. However, none of the above mentioned studies consider MapReduce jobs/tasks scheduling problem.

From algorithmic point of view: people are looking into MapReduce jobs/tasks scheduling with various considerations in different scenarios. (1) Fairness. Isard et al. [21] introduced Quincy for scheduling concurrent distributed jobs with fine-grain resource sharing and Quincy can get better fairness. Delay scheduling [37] is proposed to divide resources using max–min fair sharing to achieve statistical multiplexing. (2) Data locality. Wang et al. [33] proposed ActCap which uses a method based on Markov chain to do node-capability-aware data placement for the continuous incoming data in ever-growing heterogeneous clusters. Wang et al. [34] presented a new queueing architecture and proposed a map task scheduling algorithm constituted by the JSQ policy together with the MaxWeight policy under heavy traffic. Tan et al. [31] formulated a stochastic optimization framework to improve the data locality for reduce tasks with the optimal placement policy exhibiting a threshold-based structure. Purlieus [25] allocated VMs for MapReduce cluster in a data-locality manner to optimize performance of data access in MapReduce system. Besides, Omega [30] is proposed to support cooperation of multiple schedulers in large computer clusters. Zheng et al. [41] propose a MapReduce scheduler with provable efficiency on total flow time. Sandholm et al. [28] provided automatic application-independent optimization strategies by prioritizing users, stage in a job, and bottleneck components within a stage. However, the goals of these studies are not so much finding an optimal solution, or even understanding how close to the optimal their schemes are.

For the objective of fast completion time, two most closely related works of our work are [7,8]. In [7], Chang et al. focused on a theoretical model for determining which MapReduce jobs to schedule at what times, and formulated a linear program and several approximation algorithms like OFFA for minimizing the total job completion times. In [8], Chen et al. investigated precedence constraints between map tasks and reduce tasks in MapReduce jobs, and proposed an 8-approximation algorithm MARES, which is an advanced work and has currently the best theoretical performance upper bound, to solve the joint scheduling problem. However, they assume that tasks are assigned to processors/servers in advance. As shown in Fig. 1, scheduling of jobs without considering server assignment may result in less optimal solutions. We fill in this gap in this paper.

General scheduling of parallel machines has decades of studies. There are many works with inspiring ideas and analytical techniques [22,29,17,10]. In particular, the polyhedron of necessary conditions for the single machine problem was derived in [26]. And [29] used this result to derive approximation algorithms for single machine weighted completion time with additional side constraints. For minimizing total weighted job completion time with precedence constraints on multi-machines, the best known work is a 4-approximation algorithm in [27]. However, these works focus on the general case.

3. MapReduce server-job organizer: the problem and complexity analysis

3.1. Problem formulation

Let $\mathbb{J}$ be a set of MapReduce jobs. Let $\mathbb{M}$ be a set of identical machines. Let the release time of job j be r_j. This *release time* is the time a job is entering the system; note that it differs from the job *start time* where the job scheduler can schedule a job to be started later than this release time. Let $\mathbb{T}_j^{(M)}$ and $\mathbb{T}_j^{(R)}$ be the set of map tasks and reduce tasks for each job j. Let $\mathbb{T}$ be the set of all tasks of $\mathbb{J}$. For each task $u \in \mathbb{T}$, let p_u be its processing time. We assume that a task cannot be preempted. In our assumption, MapReduce jobs can be preempted; only tasks cannot. In the current operation of MapReduce job scheduling, a job with high priority does not preempt running tasks. It waits until the processors are released. This is because that the number of processors is large and the tasks are relatively small. Thus, the high priority job can be put into execution quickly. We also assume that for any job j, processing times of its map tasks are smaller than that of its reduce tasks. We admit that this is a key assumption for our bounding development. Yet this is true in current situation. Every map task simply scatters a chuck of data while the reduce tasks need to gather, reorganize and process data produced by map tasks. To make the situation worse, the number of reduce task is always configured to be much less than the number of map tasks. As a result, processing times of reduce tasks are much longer than that of map tasks. We also validate this assumption in our experiment.

Let d_{uv} be the delay between a map task $u \in \mathbb{T}_j^{(M)}$ and a reduce task $v \in \mathbb{T}_j^{(R)}$ (e.g., introduced by shuffle phase). Let S_u be the start time of task u. Let $\mathbb{S}$ be set of S_u, $\forall u \in \mathbb{T}$. Let C_j be completion time of job j, which is the time when all reduce tasks $v \in \mathbb{T}_j^{(R)}$ finish. Let $\mathbb{C}$ be set of C_j, $\forall j \in \mathbb{J}$. There is a weight w_j associated with job j and our objective is to find a feasible schedule to minimize total weighted job completion time $\sum_{j\in\mathbb{J}} w_j C_j$ subject to following

Table 1
Summary of key notations.

Notation	Definition	Notation	Definition
$\mathbb{J}$	Set of all jobs	S_u	Start time of task u
$\mathbb{T}$	Set of all tasks in $\mathbb{J}$	p_u	Processing time of task u
$\|\mathbb{T}\|$	Task number in $\mathbb{T}$	d_{uv}	Delay between task u and task v
$\mathbb{M}$	Set of machines	$\mathbb{S}$	Set of start times of tasks
$\|\mathbb{M}\|$	Machine number in $\mathbb{M}$	M_u	Middle finish time of task u
C_j	Completion time of job j	G	Precedence graph of all tasks
$\mathbb{C}$	Set of job completion time	$\mathbb{B}$	Any subset of all tasks set $\mathbb{T}$
w_j	Weight of job j	$\hat{C}_u$	Completion time of task u
r_j	Release time of job j	$\hat{\mathbb{C}}$	Set of task completion time
$\mathbb{T}_j^{(M)}$	Map task set of job j	$\hat{w}_u$	Weight of task u
$\mathbb{T}_j^{(R)}$	Reduce task set of job j	$\hat{r}_u$	Release time of task u

constraints for every job j. We can write the linear program model as follows (see Table 1):

$$\min \sum_{j\in\mathbb{J}} w_j C_j$$

s.t.

$$S_u \geq r_j \qquad \forall u \in \mathbb{T}_j^{(M)} \tag{1}$$

$$S_v \geq S_u + d_{uv} + p_u \qquad \forall u \in \mathbb{T}_j^{(M)}, v \in \mathbb{T}_j^{(R)} \tag{2}$$

$$C_j \geq S_v + p_v \qquad \forall v \in \mathbb{T}_j^{(R)}. \tag{3}$$

3.2. Problem complexity

Theorem 1. *MSJO is NP-complete.*

Proof. It is easy to verify that calculating total weighted job completion time of a schedule result is NP. Therefore, MSJO is in NP class. To show MSJO is NP-complete, we reduce a job schedule problem (problem SS13 in [15]) to it. Problem SS13 is proven NP-complete to determine a feasible schedule for minimizing the total weighted job completion time. Given every instance $(\mathbb{J}, \mathbb{M})$ of problem SS13, where $\mathbb{J}$ is a set of jobs and $\mathbb{M}$ is a set of identical machines. We can construct an instance $(\mathbb{J}^M, \mathbb{M}^M)$ of MSJO. $\mathbb{M}$ and $\mathbb{M}^M$ are same. For every job $j \in \mathbb{J}$, there is a job $j^M \in \mathbb{J}^M$. j and j^M have same job weight. j^M has one map task with processing time 0 and one reduce task with processing time of job j. Release time of j^M is 0. Thus, if MSJO can be solved optimally with a polynomial algorithm, problem SS13 can be solved by this algorithm. Because problem SS13 is NP-complete, MSJO is NP-complete. □

4. Algorithm development and theoretical analysis

We outline our approach described in next three subsections: (1) We introduce a linear programming relaxation to give a lower bound of the optimal solution for MSJO. This LP-based solution may not be a feasible solution. (2) Although there is a polynomial time algorithm for solving this LP relaxed problem in theory, the high complexity associated makes it impractical to solve the LP-relaxed problem when problem size is large. Therefore, inspired by this classic linear programming relaxation, we develop a novel constraint generation algorithm to produce another relaxed solution which provides lower bound to MSJO. (3) We develop algorithm MarS to generate a feasible solution from this relaxed solution. We prove that this solution is within 3 factors of the optimal solution for MSJO.

4.1. Classical linear programming relaxation

Since MSJO is NP-complete, we adopt a linear programming relaxation of the problem to give a lower bound on the optimal solution value. Constraints of this LP relaxation are necessary conditions that task start times in a feasible schedule result have to satisfy. The relaxation constraints are shown as follows:

$$\sum_{u\in\mathbb{B}} p_u S_u \geq \frac{1}{2|\mathbb{M}|}\left(\sum_{u\in\mathbb{B}} p_u\right)^2 - \frac{1}{2}\sum_{u\in\mathbb{B}} p_u^2 \qquad \forall \mathbb{B} \subseteq \mathbb{T} \tag{4}$$

where $|\mathbb{M}|$ is number of machines in $\mathbb{M}$, $\mathbb{B}$ is any subset of $\mathbb{T}$.

Then our linear programming relaxation problem is minimizing $\sum_{j\in\mathbb{J}} w_j C_j$ subjected to constraints in Eqs. (1)–(4). We call this problem Classical LP Relaxation Problem (CLS-LPP). Note that the decision variables in this CLS-LPP are S_u and C_j; so a solution can be presented as $(\mathbb{S}, \mathbb{C})$.

Constraints in Eq. (4) describe a polyhedron where task start times of a feasible schedule lie in. We give a simple example to explain the intuition. Consider 3 machines with 6 tasks $t_1, t_2, \ldots, t_6$ whose processing times are $p_1, p_2, \ldots, p_6$ respectively. Consider an assignment result where t_1 and t_2 are assigned to machine 1, t_3 and t_4 to machine 2, t_5 and t_6 to machine 3. Start times of t_1 and t_2 can be $S_1 = 0$ and $S_2 = p_1$. Or $S_1 = p_2$ and $S_2 = 0$ if t_2 is scheduled first. Then, we have $p_1S_1 + p_2S_2 = p_1p_2 = \frac{1}{2}((p_1 + p_2)^2 - (p_1^2 + p_2^2))$. Tasks on other machines have similar equations. Adding these equations together, we have $\sum_{i=1}^6 p_iS_i = \frac{1}{2}(p_1+p_2)^2+\frac{1}{2}(p_3+p_4)^2+\frac{1}{2}(p_5+p_6)^2-\frac{1}{2}\sum_{i=1}^6 p_i^2 \geq \frac{1}{2}\times\frac{1}{3}(\sum_{i=1}^6 p_i)^2-\frac{1}{2}\sum_{i=1}^6 p_i^2$ where equality holds when $p_1+p_2 = p_3 + p_4 = p_5 + p_6 = \frac{1}{3}\sum_{i=1}^6 p_i$. Note that this argument can be extended to any feasible task schedule results. When additional constraints are added, the sum of task start times will increase. As a result, the left part of Eq. (4) increases and the relation still holds.

4.2. Conditional LP relaxation and constraint generation algorithm

Note that there are an exponential number of constraints in Eq. (4) due to the exponential number of $\mathbb{B}$. To the best of our knowledge, algorithms for solving LP problems need to handle all constraints. Their computing complexities are at least $O(n)$ where n is the number of constraints in the problem because algorithms need to check whether all constraints are satisfied. In our LP-relaxed problem, the exponential number of constraints makes the computing complexity unacceptable. We derive a new LP-relaxation problem which has a small subset of constraints in Eq. (4). The optimal result of this new LP-relaxation problem also leads to the 3-approximation algorithm to be developed in Section 4.3. Because this new problem is built by iteratively adding constraints based on checking certain property of its solution, we call it Conditional LP-relaxation problem (CND-LPP).

Before developing our algorithm for building CND-LPP, we introduce a property of the solutions that satisfy Eq. (4). Given start time S_u of task u, let $M_u = S_u + \frac{1}{2}p_u$ be *middle finish time* of task u. The property is described as follows:

Property 1. *Given $\mathbb{S}$ satisfying Eq. (4), we sort tasks in non-descending order of their middle finish times. We use a permutation π to represent the sorting result, where $M_{\pi(1)} \leq M_{\pi(2)} \leq \cdots \leq M_{\pi(|\mathbb{T}|)}$. We have following inequation for all $i \in [2 \ldots |\mathbb{T}|]$:*

$$\frac{1}{|\mathbb{M}|}\sum_{k=1}^{i-1} p_{\pi(k)} \leq 2M_{\pi(i)}. \tag{5}$$

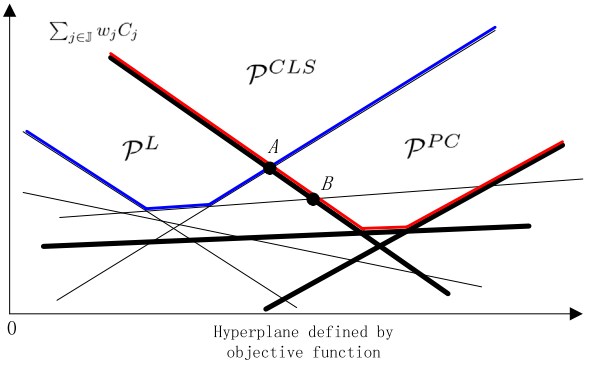

Fig. 3. Illustration of optimal solutions of CLS-LPP and CND-LPP. Fine black lines represent constraints in Eq. (4). Thick black lines represent constraints in Eqs. (1)–(3). Blue lines represent boundary of $\mathcal{P}^L$. Red lines represent boundary of $\mathcal{P}^{PC}$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Proof. For a given permutation π and task $\pi(i)$, we create a task set $\mathbb{B} = \{\pi(1), \pi(2), \ldots, \pi(i-1)\}$. We rewrite Eq. (4) as:

$$\sum_{k=1}^{i-1} p_{\pi(k)} \left(M_{\pi(k)} - \frac{p_{\pi(k)}}{2} \right) \geq \frac{1}{2|\mathbb{M}|} \left(\sum_{k=1}^{i-1} p_{\pi(k)} \right)^2 - \frac{1}{2} \sum_{k=1}^{i-1} p_{\pi(k)}^2. \quad (6)$$

Then we have

$$\sum_{k=1}^{i-1} p_{\pi(k)} M_{\pi(k)} \geq \frac{1}{2|\mathbb{M}|} \left(\sum_{k=1}^{i-1} p_{\pi(k)} \right)^2.$$

Because $M_{\pi(1)} \leq M_{\pi(2)} \leq \cdots \leq M_{\pi(i)}$, we have:

$$M_{\pi(i)} \sum_{k=1}^{i-1} p_{\pi(k)} \geq \sum_{k=1}^{i-1} p_{\pi(k)} M_{\pi(k)} \geq \frac{1}{2|\mathbb{M}|} \left(\sum_{k=1}^{i-1} p_{\pi(k)} \right)^2.$$

Finally, we have Eq. (5) by eliminating $\sum_{k=1}^{i-1} p_{\pi(k)}$. □

Given a solution of CLS-LPP, if we schedule tasks in non-descending order of their middle finish time, Property 1 gives us a basic relation between the total processing time of previous scheduled tasks and the middle finish time of the unscheduled tasks. We will use this property to prove the theoretical bound of our algorithm MarS. Note that this property holds for any solution satisfying constraints in Eq. (4). If we can build a CND-LPP whose optimal solution also has this property, MarS has the same theoretical bound based on this CND-LPP.

Recall that the intuition behind Eq. (4) is to describe a polyhedron where the task start times of a feasible schedule lie in. We denote this polyhedron as $\mathcal{P}^L$. Given a specific CLS-LPP, its constraints define a polyhedron, denoted as $\mathcal{P}^{CLS}$. All precedence constraints in Eqs. (1)–(3) define another polyhedron, denoted as $\mathcal{P}^{PC}$. We know that $\mathcal{P}^{CLS} = \mathcal{P}^L \cap \mathcal{P}^{PC}$ (see Fig. 3). Objective function of the problem defines a hyperplane. Solving CLS-LPP is searching for a point in $\mathcal{P}^{CLS}$ which has the smallest distance to this hyperplane. Instead of finding the optimal point in $\mathcal{P}^{CLS}$ (point A in Fig. 3), we search a solution in $\mathcal{P}^{PC}$ (point B in Fig. 3) which has Property 1. We start with an initial CND-LPP which only contains all precedence constraints in Eqs. (1)–(3). By iteratively adding fine chosen constraints in Eq. (4), we approach the desirable solution.

To check whether an optimal solution satisfies Property 1, we can check whether constraints in Eq. (6) are satisfied by this optimal solution for every task $\pi(i)$. We formally define these constraints as follows:

Definition. Performance Guarantee Constraint for given π and index i, denoted as $PGC(\pi, i)$, is defined as follows:

$$\sum_{k=1}^{i-1} p_{\pi(k)} S_{\pi(k)} \geq R(\pi, i)$$

where $R(\pi, i) = \frac{1}{2|\mathbb{M}|} \left(\sum_{k=1}^{i-1} p_{\pi(k)} \right)^2 - \frac{1}{2} \sum_{k=1}^{i-1} p_{\pi(k)}^2$.

We call them performance guarantee constraints because if an optimal LP satisfies $PGC(\pi, i)$ for $\forall i \in [2 \ldots |\mathbb{T}|]$, MarS can produce a feasible solution with guaranteed performance.

We first describe the main process of our constraint generation algorithm (COGE) (see Algorithm 1). First we build an initial CND-LPP. In this initial CND-LPP, all precedence constraints in Eqs. (1)–(3) are included. Its objective function is same as MSJO. Then, there are 3 main steps. In step 1, we solve CND-LPP and get an optimal solution $(\mathbb{S}^{LP}, \mathbb{C}^{LP})$ for current CND-LPP. In step 2, we use $\mathbb{S}^{LP}$ to produce π and build PGCs based on π. In step 3, we check whether $\mathbb{S}^{LP}$ satisfies $PGC(\pi, i)$ for $\forall i \in [1 \ldots |\mathbb{T}|]$. If $\mathbb{S}^{LP}$ satisfies all PGCs, we are done. Otherwise, we add the violated PGCs to CND-LPP and repeat steps 1–3 until we produce an optimal solution that satisfies all PGCs.

Because finding an optimal solution satisfying all PGCs may still involve large computation complexity in large problems, given a threshold ϵ we can terminate computation if current solution $(\mathbb{S}^{LP}, \mathbb{C}^{LP})$ is within $(1-\epsilon)$ of optimal solution satisfying all PGCs. Unfortunately, optimal solution is hard to compute. Instead, we construct a feasible solution $(\mathbb{S}^{NV}, \mathbb{C}^{NV})$ which satisfies all PGCs (NV means no-violation). When $PGC(\pi, i)$ is not satisfied, we calculate an offset which indicates how much $S_{\pi(i)}$ should increase to satisfy $PGC(\pi, i)$. Function $ViolationOffset(PGC(\pi, i), \mathbb{S}^{LP})$ calculates this offset as follows:

$$\text{offset} = \frac{1}{p_{\pi(i-1)}} \left(R(\pi, i) - \sum_{k=1}^{i-1} p_{\pi(k)} S_{\pi(k)}^{LP} \right).$$

Thus, we can build a feasible solution by adding this offset to $S_{\pi(k)}^{NV}$ for $k \in [i-1 \ldots |\mathbb{T}|]$. Finally, after all PGCs are checked, we check whether $(\mathbb{S}^{LP}, \mathbb{C}^{LP})$ reaches the stop threshold. COGE is a heuristic-based solution for the LP-relaxed problem. In practice, COGE can produce satisfactory result in less than 10 iterations.

Algorithm 1 COGE()

Input: 1) Job set $\mathbb{J}$; 2) Machine set $\mathbb{M}$; 3) Stop threshold ϵ
Output: A solution $(\mathbb{S}^{LP}, \mathbb{C}^{LP})$.

```
Build initial CND-LPP with Eq. (1)–(3);
repeat
    Solve CND-LPP with a linear programming solver;
    Let (S^LP, C^LP) be optimal solution of current CND-LPP;
    Let violated PGC number vn be 0;
    (S^NV, C^NV) ← (S^LP, C^LP);
    π ← Sort tasks by middle finish time according to S^LP;
    for all i ∈ [1 ... |T|] do
        if PGC(π, i) is satisfied by S^LP then
            continue;
        Add PGC(π, i) to CND-LPP;
        vn = vn + 1;
        offset = ViolationOffset(PGC(π, i), S^LP)
        for all k ∈ [i − 1 ... T] do
            S^NV_π(k) = S^NV_π(k) + offset;
        Update C^NV according to S^NV;
        if Σ_{j∈J} w_j C_j^LP ≥ (1 − ε) Σ_{j∈J} w_j C_j^NV then
            return (S^LP, C^LP);
until vn is 0
return (S^LP, C^LP);
```

4.3. MarS algorithm

In this section, we describe MarS. MarS is a heuristic algorithm which derives feasible schedule result from the optimal solution of the linear relaxation problem. Let $(\mathbb{S}^{LP}, \mathbb{C}^{LP})$ denote the optimal result of our LP relaxation problem. Let M_u^{LP} denote the middle finish time of task u in the LP optimal result. We have $M_u^{LP} = S_u^{LP} + p_u/2, \forall u \in \mathbb{T}$. Let $\mathbb{S}^H$ be set of task start times in final schedule result.

Our algorithm MarS is shown in Algorithm 2. We first produce π based on $\mathbb{S}^{LP}$. Then we schedule tasks from $\pi(1)$ to $\pi(|\mathbb{T}|)$, meaning we schedule tasks in non-descending order of their middle finish time. In each iteration i, we first check the earliest possible start time $S^{earliest}$ of task $\pi(i)$ to make sure that precedence constraints are satisfied. Then we choose a machine m^* which has the earliest idle time among all machines. We schedule $\pi(i)$ to machine m^* with start time $\max\{S^{earliest}, e_{m^*}\}$ and update e_{m^*} as finish time of $\pi(i)$.

Because MarS schedules tasks in non-descending order of their middle finish times, Property 1 gives us a basic relation between the total processing time of previous scheduled tasks and middle finish time of unscheduled tasks. When release time constraints and precedence constraints exist, there may be idle time intervals between tasks. Thus, we introduce Lemma 2 to give an upper bound to the total length of these intervals.

Lemma 2. *Given schedule result after tasks $\pi(k)$, $k \in [1 \ldots (i-1)]$, are scheduled by MarS, if we choose any machine m and start $\pi(i)$ as soon as possible after machine m is idle, it results in a start time $S^H_{\pi(i)}$ for $\pi(i)$. Let $g(m, \pi(i))$ be total length of idle time interval on machine m before $S^H_{\pi(i)}$. We have:*

$$g(m, \pi(i)) \le M^{LP}_{\pi(i)}. \tag{7}$$

Proof. We outline our idea first. In our schedule result, there is an idle time interval before a task because this task cannot start earlier due to certain precedence constraints. These constraints are tight in our schedule result but may not be tight in LP optimal result. For example, there is an idle time interval between tasks u_2 and v_3 in Fig. 4 because $S^H_{u_2} = S^H_{u_1} + d_{u_1u_2} + p_{u_1}$. Otherwise, task u_2 can start earlier. However, we only know $S^{LP}_{u_2} \ge S^{LP}_{u_1} + d_{u_1u_2} + p_{u_1}$. Our idea is to prove $S^H_{u_2} - (S^H_{v_3} + p_{v_3}) \le M^{LP}_{u_2} - M^{LP}_{v_3}$ by analyzing these tight precedence constraints in our schedule result. The left part of this inequation is maximum length of idle time interval between v_3 and u_2. We iteratively develop similar inequations for idle time intervals before task v_3. Sum up both sides of these inequations, we have Eq. (7).

Next, we start our formal proof. For a scheduling problem, we can build a precedence graph $G = (V, L)$ to describe all precedence constraints with delay. V is the set of tasks. For two tasks u and v, directed link $(u, v) \in L$ with length d_{uv} indicates that execution of v at least waits for a time interval d_{uv} after u finishes. For our problem, we introduce a dummy initial task t^l to represent the start point of the schedule. Then, constraints $S_u \ge r_j,\ \forall u \in \mathbb{T}_j^{(M)}$ in Eq. (1) can be expressed in the form of precedence constraint with delay: $S_u \ge S_{t^l} + d_{t^lu} + p_{t^l},\ \forall u \in \mathbb{T}_j^{(M)}$ where $S_{t^l} = 0, d_{t^lu} = r_j, p_{t^l} = 0$. In remaining part of this proof, we only mention precedence constraint with delay.

Given a schedule result, delay between u and v may be longer than d_{uv}. We say a link $(u, v) \in L$ is *tight* in this schedule result if $S^H_v = S^H_u + d_{uv} + p_u$. In a schedule result, there may be a *tight-link path* $\{u_1 \to u_2 \to \cdots \to u_s\}$ where link $(u_k, u_{k+1}) \in L$ is tight for all $k \in [1 \ldots s-1], s \ge 2$. Fig. 4 demonstrates a three-node tight-link path in a schedule result. If $u_s = u$, we call $\{u_1 \to u_2 \to \cdots \to u_s\}$ *a tight-link path of task u*. Among all tight-link paths of task u, there is a tight-link path which has the maximal number of nodes. We call it the *longest tight-link path of task u*, denoted as $LTLP(u)$. In our problem, for a map task $u \in T_j^{(M)}$, $LTLP(u)$ can be empty or $LTLP1 = \{t^l \to u\}$. For a reduce task $v \in T_j^{(R)}$, $LTLP(v)$ can be $LTLP2 = \{t^l \to u \to v\}$ or $LTLP3 = \{u \to v\}$ where $u \in T_j^{(M)}$.

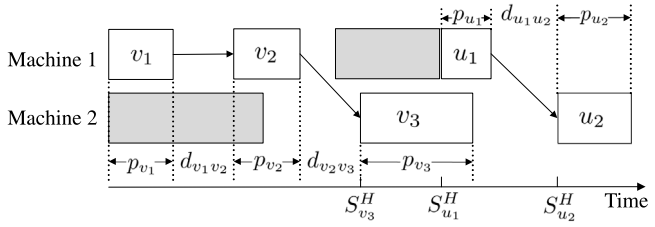

Fig. 4. Demo of tight-link path $v_1 \to v_2 \to v_3$ and $u_1 \to u_2$ in a schedule result for proof of Lemma 2.

For a tight link (u, v), the following inequation holds for the optimal result of LP relaxation problem because precedence constraints are satisfied:

$$S^{LP}_v \ge S^{LP}_u + d_{uv} + p_u.$$

Then we have:

$$M^{LP}_v \ge M^{LP}_u + d_{uv} + p_u + \frac{1}{2}(p_v - p_u).$$

For a tight-link path $\{u_1 \to u_2 \to \cdots \to u_s\}$, we have:

$$M^{LP}_{u_s} \ge M^{LP}_{u_1} + \sum_{k=1}^{s-1}(d_{u_ku_{k+1}} + p_{u_k}) + \frac{1}{2}\left(p_{u_s} - p_{u_1}\right).$$

For $LTLP1$, $LTLP2$ and $LTLP3$, we always have $p_{u_s} \ge p_{u_1}$ because in a specific job, processing times of its reduce tasks are longer than that of its map tasks. Then we have:

$$M^{LP}_{u_s} \ge M^{LP}_{u_1} + \sum_{k=1}^{s-1}(d_{u_ku_{k+1}} + p_{u_k}). \tag{8}$$

Because all links in a tight link path are tight, we also have:

$$S^H_{u_s} - S^H_{u_1} = \sum_{k=1}^{s-1}(d_{u_ku_{k+1}} + p_{u_k}). \tag{9}$$

Based on Eqs. (8) and (9), we analyze lengths of idle time intervals on machine m. After scheduling $\pi(i)$ to machine h, there are idle time intervals before $S^H_{\pi(i)}$. Considering a task $\hat{u}$ right after a idle time interval, we find $LTLP(\hat{u}) = \{u_1 \to u_2 \to \cdots \to u_s\}$. Because u_1 is the first task in $LTLP(u_s)$, there must be a task $\hat{v}$ scheduled on machine m where $S_{\hat{v}} \le S_{u_1} \le (S_{\hat{v}} + p_{\hat{v}})$ (see task v_3 in Fig. 4) and $M^{LP}_{\hat{v}} \le M^{LP}_{u_1}$. Otherwise, u_1 can start earlier. With Eqs. (8) and (9), we have:

$$S_{u_s} - (S_{\hat{v}} + p_{\hat{v}}) \le M^{LP}_{u_s} - M^{LP}_{\hat{v}}.$$

The left side of this inequation is the maximum length of idle time interval between $\hat{u}$ and $\hat{v}$. We repeat developing this inequation for idle time interval before $\hat{v}$. Finally, we end up with t^l whose middle finish time is 0. By adding all these inequations together, we have Eq. (7). □

Lemma 2 gives an upper bound for total idle time intervals in the scheduling of task $\pi(i)$. Note that, this result only holds when tasks from $\pi(1)$ to $\pi(i-1)$ are scheduled according to MarS. Next, we introduce Theorem 3.

Algorithm 2 MarS()

Input: 1) Job set $\mathbb{J}$; 2) Machine set $\mathbb{M}$; 3) LP optimal result $\mathbb{S}^{LP}$
Output: Scheduling result $\mathbb{S}^{H}$.

$\mathbb{S}^{H} \leftarrow \emptyset$;
$\pi \leftarrow$ Sort tasks by middle finish time according to $\mathbb{S}^{LP}$;
Let e_m be earliest idle time of machines m;
$e_m \leftarrow 0, \forall m \in \mathbb{M}$;
for all $i \in [1 \dots |\mathbb{T}|]$ **do**
  Find job j where $\pi(i) \in \mathbb{T}_j^{(M)}$ or $\pi(i) \in \mathbb{T}_j^{(R)}$;
  if $\pi(i) \in \mathbb{T}_j^{(M)}$ **then**
    $S^{earlist} = r_j$;
  if $\pi(i) \in \mathbb{T}_j^{R}$ **then**
    $S^{earlist} = \max_{u \in \mathbb{T}_j^{(M)}, S_u^H \in \mathbb{S}^H}\{S_u^H + p_u + d_{u\pi(i)}\}$;
  Find m^* where $e_{m^*} = \min_{m \in \mathbb{M}}\{e_m\}$;
  $S_{\pi(i)}^H = \max\{S^{earlist}, e_{m^*}\}$;
  $e_{m^*} = S_{\pi(i)}^H + p_{\pi(i)}$;
  $\mathbb{S}^H \leftarrow \mathbb{S}^H \bigcup S_{\pi(i)}^H$;
return $\mathbb{S}^H$;

Theorem 3. *MarS is a 3-approximation algorithm for MSJO.*

Proof. Consider $\pi(i)$ is scheduled to machine m. Let $\mathbb{U}(m, \pi(i))$ be set of tasks scheduled to machine m before $\pi(i)$. Combine Eqs. (5) and (7), we have:

$$\frac{1}{|\mathbb{M}|}\sum_{k=1}^{i-1} p_{\pi(k)} + g(m, \pi(i)) \le 3M_{\pi(i)}^{LP}.$$

In our algorithm, we choose machine to start task $\pi(i)$ as early as possible, so we have:

$$S_{\pi(i)}^H \le \sum_{u \in \mathbb{U}(m,\pi(i))} p_u + g(m, \pi(i)), \quad \forall m \in \mathbb{M}.$$

There must exist a machine $\hat{m}$ where $\sum_{u \in \mathbb{U}(\hat{m},\pi(i))} p_u \le \frac{1}{|\mathbb{M}|}\sum_{k=1}^{i-1} p_{\pi(k)}$. Then, $S_{\pi(i)}^H \le 3M_{\pi(i)}^{LP}$ and we have:

$$S_{\pi(i)}^H + p_{\pi(i)} \le 3M_{\pi(i)}^{LP} + \frac{3}{2}p_{\pi(i)} = 3(S_{\pi(i)}^{LP} + p_{\pi(i)}). \tag{10}$$

Eq. (10) holds for all task $\pi(i), i \in [1 \dots |\mathbb{T}|]$, then $C_j^H \le 3C_j^{LP}$, $\forall j \in J$. Finally, we have:

$$\sum_{j \in \mathbb{J}} w_j C_j^H \le 3\sum_{j \in \mathbb{J}} w_j C_j^{LP}.$$

Because $\sum_{j \in \mathbb{J}} w_j C_j^{LP}$ is a lower bound of the optimal, MarS is a 3-approximation algorithm for MSJO. □

4.4. Extension: a fine-grained MapReduce task scheduling algorithm(T-MarS)

In this section, we propose a fine-grained task-level MapReduce scheduler, that means our optimization objective is to minimize $\sum_{u \in \mathbb{T}} \hat{w}_u \hat{C}_u$ ($\hat{w}_u$ denotes the weight of task u and $\hat{C}_u$ denotes the completion time of task u, note that $\hat{C}_u = S_u + p_u$). Generally, instead of considering the coarse-grained submission time of a job, we assume every task has a release time and let $\hat{r}_u$ denote the release time of task u, which indicates the earliest time that the task can be scheduled. We can rewrite the LP model as follows:

$$\min \sum_{u \in \mathbb{T}} \hat{w}_u \hat{C}_u$$

s.t.

$$\hat{C}_u \ge \hat{r}_u + p_u \qquad \forall u \in \mathbb{T} \tag{11}$$

$$\hat{C}_v \ge \hat{C}_u + p_u \qquad \forall (u, v) \in L. \tag{12}$$

Apparently, according to the previous proof in Section 3.2, we can use the similar method to prove that it is also a NP-complete problem. For minimizing total weighted job completion time with precedence constraints, the best known work is an 8-approximation algorithm in [8] without considering the impact of processors/machines assignment. Here, we develop a fast 7-approximation heuristic algorithm T-MarS.

Before developing our heuristic algorithm, we first adopt a linear programming relaxation of the NP-hard problem to give a lower bound on the optimal solution value, which are necessary conditions that task completion times in a feasible schedule result have to satisfy. The relaxation constraints are shown as follows:

$$\sum_{u \in \mathbb{B}} p_u \hat{C}_u \ge \frac{1}{2|\mathbb{M}|}\left(\sum_{u \in \mathbb{B}} p_u^2 + \left(\sum_{u \in \mathbb{B}} p_u\right)^2\right) \qquad \forall \mathbb{B} \subseteq \mathbb{T} \tag{13}$$

where $|\mathbb{M}|$ is number of machines in $\mathbb{M}$, $\mathbb{B}$ is any subset of $\mathbb{T}$.

Proof. To prove Eq. (13) we give a brief explanation. Consider a subset $\mathbb{I} \subseteq \mathbb{T}$, we sort tasks in non-descending order of their completion times. Let $\mathbb{I} = \{1, 2, \dots, u\}$, then task u is the last task to be finished. Assume the performances of all the processors/machines are identical, if task u is scheduled on machine m^* ($m^* \in \mathbb{M}$), then m^* is the most heavily loaded machine. So the load on machine m^* is at least $(\sum_{i \in \mathbb{I}} p_i)/|\mathbb{M}|$, and $\hat{C}_u \ge (\sum_{i \in \mathbb{I}} p_i)/|\mathbb{M}|$, we have:

$$\sum_{u \in \mathbb{T}} p_u \hat{C}_u \ge \frac{1}{|\mathbb{M}|}\sum_{u \in \mathbb{T}}\left(p_u \sum_{i \in \mathbb{I}} p_i\right).$$

Because $\frac{1}{|\mathbb{M}|}\sum_{u \in \mathbb{T}}(p_u \sum_{i \in \mathbb{I}} p_i) = \frac{1}{|\mathbb{M}|}\sum_{u \in \mathbb{T}}\sum_{i \in \mathbb{I}} p_i p_u = \frac{1}{2|\mathbb{M}|}(\sum_{u \in \mathbb{T}} p_u^2 + (\sum_{u \in \mathbb{T}} p_u)^2)$, then we have:

$$\sum_{u \in \mathbb{T}} p_u \hat{C}_u \ge \frac{1}{2|\mathbb{M}|}\left(\sum_{u \in \mathbb{T}} p_u^2 + \left(\sum_{u \in \mathbb{T}} p_u\right)^2\right).$$

Therefore, when $\mathbb{B} = \mathbb{T}$ Eq. (13) has been proved correct. Note that this argument can be extended to the general case, so when $\mathbb{B}$ is any subset of $\mathbb{T}$, the formula is always workable. □

Next, we introduce a property of the solutions that satisfy Eq. (13). The property is described as follows:

Property 2. *Given $\hat{C}_1, \hat{C}_2, \dots, \hat{C}_{|\mathbb{T}|}$ satisfying Eq. (13), we sort tasks in non-descending order of their completion times, without loss of generality, assume $\hat{C}_1 \le \hat{C}_2 \le \cdots \le \hat{C}_{|\mathbb{T}|}$. Let $\mathbb{B} = \{1, 2, \dots, i\}$ and $j \in \mathbb{B}$, then we have:*

$$\hat{C}_i \ge \frac{1}{2|\mathbb{M}|}\sum_{j \in \mathbb{B}} p_j. \tag{14}$$

Proof. Because $\hat{C}_1, \hat{C}_2, \dots, \hat{C}_{|\mathbb{T}|}$ satisfy Eq. (13), we have:

$$\sum_{j \in \mathbb{B}} p_j \hat{C}_j \ge \frac{1}{2|\mathbb{M}|}\left(\sum_{j \in \mathbb{B}} p_j^2 + \left(\sum_{j \in \mathbb{B}} p_j\right)^2\right) \ge \frac{1}{2|\mathbb{M}|}\left(\sum_{j \in \mathbb{B}} p_j\right)^2.$$

According to the hypothesis, task j is finished before task i, obviously, we have $\hat{C}_j \le \hat{C}_i$, then

$$\hat{C}_i \sum_{j \in \mathbb{B}} p_j \ge \sum_{j \in \mathbb{B}} p_j \hat{C}_j \ge \frac{1}{2|\mathbb{M}|}\left(\sum_{j \in \mathbb{B}} p_j\right)^2.$$

Consequently, we have $\hat{C}_i \ge \frac{1}{2|\mathbb{M}|}\sum_{j \in \mathbb{B}} p_j$. □

Let $(\mathbb{S}^{LP}, \hat{\mathbb{C}}^{LP})$ denote the optimal result of our LP relaxation problem. Let $\hat{\mathbb{S}}^{H'}$ be set of task start times in final schedule result. Our algorithm T-MarS is shown in Algorithm 3. First, we produce $\hat{\pi}$ based on $\hat{\mathbb{C}}^{LP}$. Second, note that the upper bound on the length of any feasible scheduling without unforced idle time is $\max_{u\in\mathbb{T}}\{\hat{r}_u\} + \sum_{u\in\mathbb{T}} p_u$, we divide the time line into intervals: $[1, 1], (1, 2], (2, 4], \ldots, (2^{N-2}, 2^{N-1}]$, where N is the smallest integer such that 2^{N-1} is at least $\max_{u\in\mathbb{T}}\{\hat{r}_u\} + \sum_{u\in\mathbb{T}} p_u$. Let $t_0 = 1$ and $t_n = 2^{n-1}$, $n \in [1, N]$, we use $\mathbb{B}_n$ to denote all the tasks which $\hat{C}^{LP}_{\hat{\pi}(i)}$ lie in interval $(t_{n-1}, t_n]$, that means $t_{n-1} \le \hat{C}^{LP}_{\hat{\pi}(i)} \le t_n$. Third, we define α_n as the average load on a machine for all the tasks in $\mathbb{B}_n$, we have $\alpha_n = (\sum_{u\in\mathbb{B}_n} p_u)/|\mathbb{M}|$. Let $t'_n = 1 + \sum_{k=1}^{n}(t_k + \alpha_k)$, for any $\mathbb{B}_n$, $n \in [1, N]$, we schedule the tasks which belong to $\mathbb{B}_n$ using Graham list-scheduling algorithm [16] in the interval $(t'_{n-1}, t'_n]$. The main idea of Graham algorithm can be stated as follows: All the tasks (without release times or release time are zero) are ordered in some list, whenever one machine becomes idle, the next available task (where a task is available if all its predecessors have been finished) on the list is allocated to start on the machine. Let $S_j^{available}$ denote the start time of next available task j and $pre(j)$ be its precedence parent node.

Algorithm 3 T-MarS()

Input: 1) Task set $\mathbb{T}$; 2) Machine set $\mathbb{M}$; 3) LP optimal result $\hat{\mathbb{C}}^{LP}$
Output: Scheduling result $\hat{\mathbb{S}}^{H'}$.

$\hat{\mathbb{S}}^{H'} \leftarrow \emptyset$;
$\hat{\pi} \leftarrow$ Sort tasks by middle finish time according to $\hat{\mathbb{C}}^{LP}$;
Let e_m be earliest idle time of machines m;
$e_m \leftarrow 0, \forall m \in \mathbb{M}$;
$N \leftarrow \lceil \max_{u\in\mathbb{T}}\{\hat{r}_u\} + \sum_{u\in\mathbb{T}} p_u \rceil$;
$\mathbb{B}_n \leftarrow 0$, $t_n \leftarrow 2^{n-1}, \forall n \in [1, ..., N]$;
for all $i \in [1 \ldots |\mathbb{T}|]$ **do**
    $n = \lceil log_2 \hat{C}^{LP}_{\hat{\pi}(i)} \rceil + 1$;
    $\mathbb{B}_n = \mathbb{B}_n \bigcup \hat{\pi}(i)$;
for all $n \in [1 \ldots N]$ **do**
    if $\mathbb{B}_n \neq \emptyset$ **then**
        $\alpha_n = (\sum_{u\in\mathbb{B}_n} p_u)/|\mathbb{M}|$;
        $t'_n = 1 + \sum_{k=1}^{n}(t_k + \alpha_k)$;
        GRAHAM($\mathbb{B}_n, t'_{n-1}, t'_n, \mathbb{M}, \hat{\mathbb{S}}^{H'}$);
return $\hat{\mathbb{S}}^{H'}$;

function GRAHAM($\mathbb{B}_n, t'_{n-1}, t'_n, \mathbb{M}, \hat{\mathbb{S}}^{H'}$)
    for all $j \in [1 \ldots |\mathbb{B}_n|]$ **do**
        $e_{m^*} = \min_{m\in\mathbb{M}}\{e_m\}$;
        $S_j^{available} = \max_{S_{pre(j)}\in\hat{\mathbb{S}}^{H'}}\{S_{pre(j)} + p_{pre(j)}\}$;
        $S_j^{H'} = \max\{S_j^{available}, e_{m^*}, t'_{n-1}\}$;
        $e_{m^*} = S_j^{H'} + p_j$;
        $\hat{\mathbb{S}}^{H'} \leftarrow \hat{\mathbb{S}}^{H'} \bigcup S_j^{H'}$;
    return;

Lemma 4. *Let $\mathbb{B}$ be the tasks subset of $\mathbb{T}$. We use Graham algorithm to schedule the tasks in $\mathbb{B}$. Let Θ denote the set of tasks that form the longest chain of precedence-constrained tasks ending with the task that completes last in the schedule. Let $\hat{C}_{\max}$ denote the maximum time length of the resulting scheduling, we have:*

$$\hat{C}_{\max} \le \frac{1}{|\mathbb{M}|}\sum_{u\in(\mathbb{B}-\Theta)} p_u + \sum_{u\in\Theta} p_u. \tag{15}$$

Obviously, we can use the pigeonhole principle to prove Lemma 4. Through the above description and analysis on Algorithm 3, we note that: (1) regularly dividing the time line different intervals can guarantee tasks scheduling relative independence; (2) in each interval scheduling the tasks using Graham algorithm can guarantee performance (a theoretical tight upper bound). Next, by applying both Property 2 and Lemma 4 we derive Theorem 5.

Theorem 5. *T-MarS is a 7-approximation algorithm for minimizing total weighted task completion times.*

Proof. We first show that Algorithm 3 can give a feasible scheduling result. Eq. (12) ensures that the precedence constraints are enforced, since for each task $i \in \mathbb{B}_n$, each of its predecessors is assigned to $\mathbb{B}_k$ for some $k \in \{1, 2, \ldots, n\}$. We also need to show that the schedule respects the release time constraints. For any $i \in \mathbb{B}_n$ and $n \in \{1, 2, \ldots, N\}$, we have:

$$\hat{r}_i \le \hat{C}_i^{LP} \le t_n.$$

According to the definition of t'_n above, we can write

$$t'_{n-1} = 1 + \sum_{k=1}^{n-1}(t_k + \alpha_k) \ge 1 + \sum_{k=1}^{n-1} t_k = t_n.$$

Hence $\hat{r}_i \le t'_{n-1}$, it means that the scheduling rule in each interval $(t'_{n-1}, t'_n]$ reduces to the case without release times. According to Lemma 4, it implies that the length of the schedule constructed for $\mathbb{B}_n$ can be bounded by the average load on each machine, plus the maximum length of any precedence chain. The average load in any interval is exactly α_n. Let β_i denote the maximum length of a chain that ends with task i. Obviously, β_i is at most $\hat{C}_i^{LP}$, that means $\beta_i \le \hat{C}_i^{LP}$. Therefore,

$$\hat{C}_i^{H'} \le t'_{n-1} + \alpha_n + \beta_i = 1 + \sum_{k=1}^{n-1}(t_k + \alpha_k) + \alpha_n + \beta_i.$$

Because $1 + \sum_{k=1}^{n-1}(t_k + \alpha_k) + \alpha_n + \beta_i = 1 + \sum_{k=1}^{n-1} t_k + \sum_{k=1}^{n}\alpha_k + \beta_i$ and $1 + \sum_{k=1}^{n-1} t_k = t_n$, then we have

$$\hat{C}_i^{H'} \le t_n + \sum_{k=1}^{n}\alpha_k + \beta_i \le t_n + \sum_{k=1}^{n}\alpha_k + \hat{C}_i^{LP}.$$

According to the definition of α_n, we can rewrite $\sum_{k=1}^{n}\alpha_k$ as follows:

$$\sum_{k=1}^{n}\alpha_k = \sum_{k=1}^{n}\left(\frac{1}{|\mathbb{M}|}\sum_{u\in\mathbb{B}_k} p_u\right) = \frac{1}{|\mathbb{M}|}\sum_{u\in(\mathbb{B}_1\cup\mathbb{B}_2\cup\cdots\cup\mathbb{B}_n)} p_u.$$

Let $\hat{C}^{LP}_{i(n)}$ denote the largest value whose $i(n) \in \mathbb{B}_1 \cup \mathbb{B}_2 \cup \cdots \cup \mathbb{B}_n$. Applying Property 2, we have

$$\frac{1}{|\mathbb{M}|}\sum_{u\in(\mathbb{B}_1\cup\mathbb{B}_2\cup\cdots\cup\mathbb{B}_n)} p_u \le 2\hat{C}^{LP}_{i(n)} \le 2t_n.$$

Since $t_{n-1} \le \hat{C}_i^{LP}$ and $2t_{n-1} = t_n$, then $t_n \le 2\hat{C}_i^{LP}$. Thus,

$$\hat{C}_i^{H'} \le t_n + 2t_n + \hat{C}_i^{LP} \le 2\hat{C}_i^{LP} + 4\hat{C}_i^{LP} + \hat{C}_i^{LP} = 7\hat{C}_i^{LP}.$$

This result holds for all task $i \in \mathbb{T}$, finally, we have

$$\sum_{i\in\mathbb{T}} \hat{w}_i\hat{C}_i^{H'} \le 7\sum_{i\in\mathbb{T}} \hat{w}_i\hat{C}_i^{LP}. \tag{16}$$

Because $\sum_{i\in\mathbb{T}} \hat{w}_i\hat{C}_i^{LP}$ is a lower bound of the optimal, T-MarS is a 7-approximation algorithm for minimizing total weighted task completion times. □

5. Simulation

5.1. Simulation setup

5.1.1. Background

We use synthetic workloads to study the performance of our algorithm, following similar simulation setup in [7,8]. We generate

jobs as follows: (1) Job release times are randomly generated following Bernoulli with probability $\frac{1}{2}$. (2) Number of tasks in a job are generated (a) uniformly or (b) randomly. For a job with uniformly generated tasks, the number of map tasks is set to 30 and the number of reduce tasks is set to 10. For a job with randomly generated tasks, the number of map tasks follows Poisson distribution with a mean of 30 and the number of reduce tasks is uniformly chosen between 1 and the number of map tasks. (3) Task processing times are generated (a) uniformly or (b) randomly. For the tasks with uniform processing time, processing times of map tasks are 10 and processing times of reduce tasks are 15. For the tasks with random processing time, processing times of map tasks are normally distributed with a mean of 10 and a standard deviation of 5. Processing times of reduce tasks are normally distributed with a mean 15 and a standard deviation 5. (4) Weights of jobs/tasks are generated randomly in normal distribution with a mean 30 and standard deviation 10. (5) Delays between map tasks and reduce tasks are proportional to the processing time of map tasks. This indicates that a long map task will generate more data and these data will need longer time to be transmitted to the reduce tasks. The default number of machines is set to 50.

5.1.2. Evaluation metric

The key to compare different algorithms is total weighted job completion time (TWJCT) of the result of the algorithms. To make comparison under different configurations more illustrative, we would like to compare TWJCT of different algorithms with the optimal solution. However, computing the optimal solutions requires exponential time. Therefore, we use the LP lower bound as a substitute. We define TWJCT ratio as our evaluation metrics:

$$\text{TWJCT ratio} = \frac{\text{TWJCT of Algorithm X result}}{\text{LP lower bound}}.$$

TWJCT ratio indicates how close the schedule result is to the theoretical lower bound. The smaller the TWJCT ratio is, the better the schedule result is. All results in our simulation are measured by TWJCT ratio.

Similarly, to compare performance of different algorithms designed for minimizing total weighted task completion times (TWTCT), we use TWTCT ratio as our evaluation metric:

$$\text{TWTCT ratio} = \frac{\text{TWTCT of Algorithm X result}}{\text{LP lower bound}}.$$

5.1.3. Comparisons strategies

We compare performance of our algorithms with the following scheduling strategies:

MARES: MARES [8] is a LP-based algorithm considering precedence constraints in MapReduce jobs. In evaluation of [8], MARES outperforms other algorithms with a factor of 1.5 to the lower bound. To offer a fair comparison, we adopt a workload-based assignment strategy where tasks are evenly allocated in order to balance total processing time of tasks on every processor. According to [29], when there is not release time and precedence constraints, LP relaxation constraints in Eq. (4) and constraints in [8] have same lower bound if workloads on every processor are same. Moreover, workload-based assignment strategy is also widely adopted in practice.

High unit weight first (HUWF): Unit weight (UW) of a job is calculated by dividing weight of the job by total processing time of tasks in the job. All tasks are sorted in descending order of unit weights of the jobs they belong to. We also maintain an available task list where all tasks in the list do not have any unscheduled precedent task. In each iteration, we choose the task with highest unit weight and assign it to a machine where it can start as early as possible. Then we check whether there is any unscheduled task whose precedent tasks are all scheduled and put these tasks into available task list. We iterate until all tasks are scheduled.

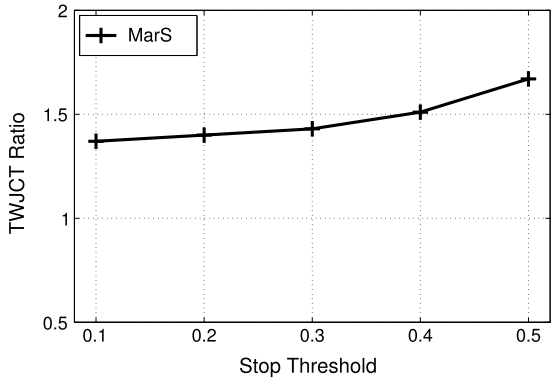

Fig. 5. Stop threshold vs. TWJCT ratio.

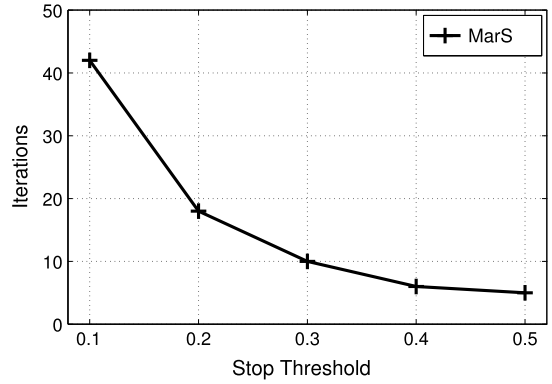

Fig. 6. Stop threshold vs. iteration number.

High job weight first (HJWF): This algorithm works similar to HUWF. The difference is that tasks are sorted according to weight of the jobs they belong to.

In the fine-grained task-level scenario, we add the following three scheduling strategies to compare their performance with T-MarS:

H-MARES: A simple heuristic implementation of MARES which we call H-MARES schedules tasks in the order of LP completion time without waiting for it to become available. The only reason for a task to wait is if some of its predecessors have not been completed. In H-MARES, it is assumed that tasks are assigned to machines in advance. We use H-MARES to evaluate effect of release time and precedence constraints.

Shortest task first (STF): When a machine is idle, we consider the available tasks set where all tasks have arrived and their predecessors have been finished, then we assign the task whose processing time is the shortest to the idle machine first. This strategy appears to be a reasonable greedy algorithm minimizing the completion time, so we take it into account.

High task weight first (HTWF): Similar to HJWF algorithm, the difference is that tasks are sorted according to their own weights.

In MarS, we need to choose a stop threshold ϵ for COGE. We run MarS with 100 jobs and change value of stop thresholds (see Figs. 5 and 6). We see that when ϵ changes from 0.1 to 0.5, there is a small performance degradation for MarS while iteration number decreases rapidly. Thus, we choose $\epsilon = 0.3$. MARES also needs a stop threshold in solving its LP-relaxation problem. To offer a fair comparison, we choose $\epsilon = 0.3$ for MARES instead of 0.5 in the original paper [8].

In our simulation, we assume that the task processing time and delays between a map task and a reduce task are known to the

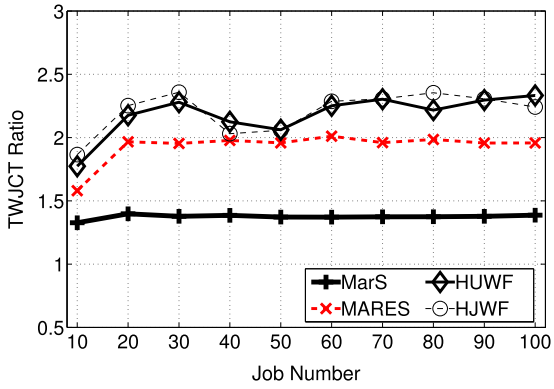

Fig. 7. Randomized scenario.

Fig. 8. Uniform task number.

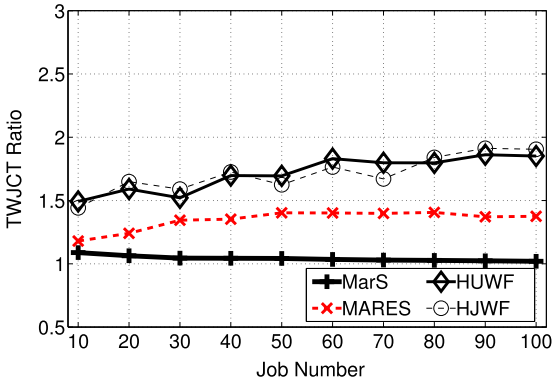

Fig. 9. Uniform task processing time.

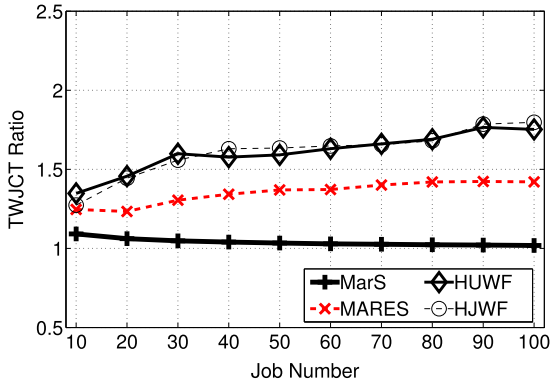

Fig. 10. Impact of machine number.

scheduler. There are many studies on accurate processing time prediction [32,36] and trace study [6] shows that the majority of map tasks and reduce tasks are highly recurrent making prediction feasible. We plan a future work here.

5.2. Simulation result

We first discuss the most randomized scenario where all parameters are generated in randomly. Based on this scenario, we evaluate impacts of different job parameters when our optimization objective is to minimize total weighted job completion times.

Performances in the most randomized scenario. Fig. 7 shows results where all job parameters are in random category. We see MarS constantly offers efficient solutions when job number changes. In theory, we proved that MarS represents a 3-approximation of the theoretical lower bound, meaning that TWJCT ratios of MarS are at most 3. In practice, MarS represents an increase less than 0.4 in terms of TWJCT ratio, compared with theoretical lower bound. More specifically, starting at 1.32, MarS increases to 1.39 when job number is 20 and stays at about 1.38 with a variance less than 0.02 when job number further grows. This is because LP relaxation always produces an optimized task schedule order. MARES shows stable performance after job number reaches 20. However, we see a constant performance difference between MARES and MarS. We consider it as improvement by joint scheduling.

We see that MarS outperforms other algorithms by over 40% in most cases. The only exception happens when job number is 10 where MarS is 1.32 while MARES is 1.58, HUWF is 1.77 and HJWF is 1.86. After job number rises to 20, MARES increases to 1.96, HUWF and HJWF jump to 2.17 and 2.25 respectively. This is because when there are less jobs, machines are not extensive loaded. Thus, different schedule algorithms can gain close performances. However, when job number increases and machines are fully utilized, algorithm results differ from each other. We also notice that MARES outperforms HUWF and HJWF in all cases. This is because workload-based allocation performs well and MARES benefits from optimized task order generated by LP relaxation conditions.

Impact of task number in job. Next, we examine the effect of task number in a job. We generate jobs with uniform task number category while other job parameters are in random category. The results are shown in Fig. 8. Compared with results in Fig. 7, we see that all algorithms gain better performance. MarS stays at about 1.15 which is 0.17 lower in terms of TWJCT ratio. MARES gradually increases from 1.48 to 1.88. HUWF and HJWF still have larger performance variance but TWJCT ratio of both algorithms decreases by 0.2 on average. It is also shown that there is only a tiny performance difference between HUWF and HJWF.

Impact of task processing time. We generate jobs with uniform task processing time category while other job parameters are in random category. MarS is very close to the theoretical lower bound. Its maximum distance to the lower bound is 0.08 when job number is 10. When job number rises, this distance decreases to less than 0.02. The gap between MarS and MARES is reduced to 0.35 on average.

It is worth noticing that comparing results in Figs. 8 and 9, all algorithms gain better performance in uniform task processing time category. This result indicates that it is more effective to have uniform task processing time than to have same task number in all jobs. It also shows the importance of solving skewed task processing time problem in a parallel computing framework.

Impact of machine number. We change machine number to 100 and examine all algorithms in most randomized scenario (see Fig. 10). We see that MarS performs extremely well. Starting at 1.09 when job number is 10, its TWJCT ratio gradually declines

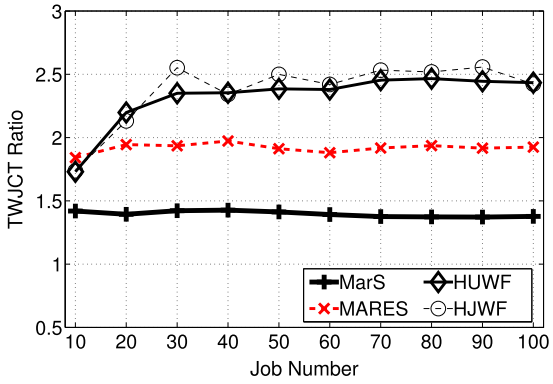

Fig. 11. Increasing UW category.

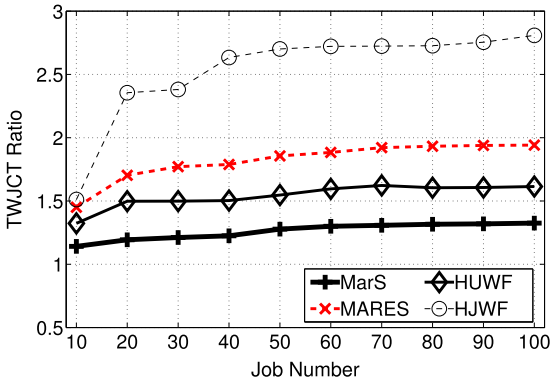

Fig. 12. Decreasing UW category.

to 1.01 which can be considered as optimal. Other algorithms also gain better performance than same scenario when machine number is 50 (see Fig. 7). Different from MarS, TWJCT ratios of other algorithms increase with job number. When job number reaches 100, MarS outperform MARES, HUWF and HJWF by 0.42, 0.75 and 0.79 respectively.

Impact of job weight. We investigate impact of job weight by generating jobs with different unit weight distribution. We first generate jobs with increasing UW category (see Fig. 11). We see that MarS and MARES do not have obvious change while both HUWF and HJWF show a 10% performance degradation. This is because in increasing UW category, later-released jobs have higher unit weight than jobs released earlier. Intuitively, tasks in later jobs should preempt the processing of tasks in earlier jobs. In order to gain better TWJCT ratio, sequence of tasks must be planned more carefully. Otherwise, a performance degradation is introduced.

Next, we generate jobs with decreasing UW category (see Fig. 12). Performances of MarS and MARES are similar in Figs. 7 and 12. However, HUWF and HJWF show opposite trends. HUWF outperforms MARES while HJWF suffers a performance degradation. Because schedule order of tasks in HUWF is same to release time order of their jobs, HUWF does not need to schedule tasks in later-released jobs to preempt tasks in earlier-released jobs while HJWF needs to take care of this because later-released job may have higher weight due to total processing time of its tasks.

Impact of prediction error. In order to investigate impact of prediction error, we inject errors to precessing times of tasks. All algorithms schedule jobs with error-injected information while we calculate a LP-lower bound based on no-error information. The result is shown in Fig. 14. *x*-axis is maximum prediction error to no-error precessing time. We see that all algorithms do not show grade performance degradation. TWJCT ratios of four algorithms increase slightly. However the maximum performance degradation is a 0.09 performance degradation in terms of TWJCT ratio when maximum prediction error is 100%.

Next, in the fine-grained task-level MapReduce scheduling scenario, which means the optimization objective is to minimize total weighted task completion times, we further compare our T-MarS scheduler with other algorithms.

Varying total task number. To evaluate the impact of total task number on performance change under different algorithms, we vary the total task number from 100 to 1000 and we generate tasks randomly each time. The results are shown in Fig. 15. We note that when total task number is 100, TWTCT ratios of T-MarS, H-MARES, STF and HTWF are 1.03, 1.19, 1.31 and 1.42 respectively, T-MarS is 0.16 less than H-MARES in terms of TWTCT ratio. However, with the growth of total task number, T-MarS is at most 1.21, while H-MARES increases to 1.6 when task number is 1000, STF and HTWF are both more than 2.2. We observe that T-MarS is closer to the theoretical lower bound than other algorithms practically, because it has strict performance guarantee constraints.

Relative waiting time. We define the waiting time of a task to be the time interval between the submission and completion time. And the relative waiting time is the ratio of the task waiting time to the longest waiting time. As shown in Fig. 16, particularly, we note that T-MarS finishes 71% of all tasks at median relative waiting time, whereas H-MARES, STF and HTWF only finish 62%, 58% and 51% respectively. The results also explain the reason that T-MarS outperforms other classical algorithms from a different perspective.

5.3. Discussion: semi-online MarS algorithm

In our above assumptions, the release time (arrival time) of all the jobs is known in advance. This is clearly not true in practice where the jobs arrive one at a time, and the arrival times cannot be known in advance in the running MapReduce platforms. In the single processor case there are some competitive algorithms for scheduling jobs online. These algorithms use the fact that scheduling the job with the highest proportion of weight to processing time can minimize the weighted completion time when all jobs are available at time zero. This is not true for our problem. However, to make our algorithm into an online version, we can use the approach as following: During a batch interval time, when a new job arrives into the system, we take all the jobs currently in the system along with the new job, when the batch interval is finished, we get all the arrival jobs' information and run MarS scheduler to give a solution. In the next batch interval, the schedule can be implemented in the same manner. We refer to this approach as Semi-Online MarS Algorithm (SO-MarS). Since we have designed a fast heuristic algorithm and run the LP model periodically, as long as we set an appropriate batch interval, the overhead is not big. To demonstrate this approach works extremely well and is very robust to varying job characteristics, we compare our algorithm with other strategies using the same simulation setup in Section 5.1. Besides, the batch interval time is set from 10 to 100 s. As shown in Fig. 17, we can see that when the batch interval is 50 s, the average TWJCT ratios of SO-MarS, MARES, HUWF and HJWF are 1.09, 1.35, 1.63, 1.75 respectively. But when the batch interval increases to 100 s, the average TWJCT ratio of SO-MarS is still less than 1.19, while that of other strategies is over 1.53. According to the results, we conclude that our Semi-Online MarS algorithm still outperforms other classical algorithms.

6. Implementation and results of experiment

6.1. Implementation

We implement a MSJO framework in Hadoop-1.2.0 and run the implementation on Amazon EC2. The implementation framework

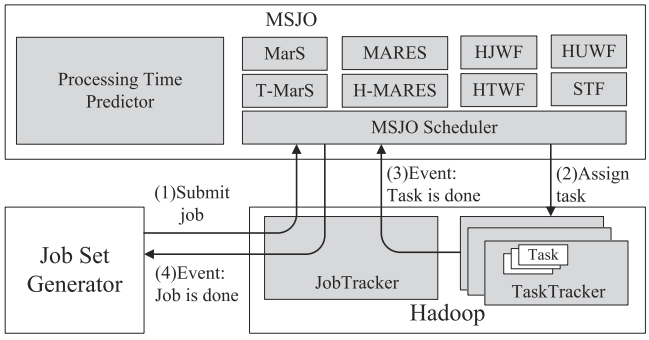

Fig. 13. Processing of a job in MarS implement with Hadoop.

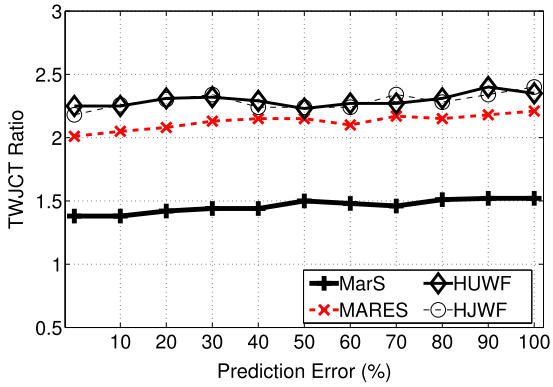

Fig. 14. Impact of prediction error.

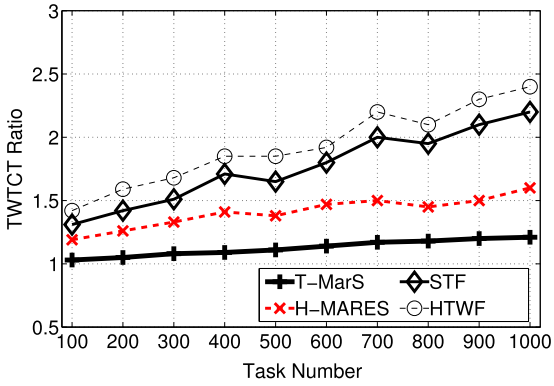

Fig. 15. Impact of total task number.

is described in Fig. 13. To run in Hadoop, MSJO needs to cooperate with two components of Hadoop: JobTracker and TaskTracker. JobTracker manages all jobs in a Hadoop cluster and, as jobs are split into tasks, TaskTracker is used to manage tasks on every machine.

We register MSJO to JobTracker so that JobTracker can call MSJO to make schedule decisions. MSJO makes schedule decisions according to different algorithm modules. Currently, we implemented four job-level algorithm modules including MarS, MARES, HUWF, HJWF, and four task-level algorithm modules including T-MarS, H-MARES, STF, HTWF in our experiments. Of course, other algorithm modules can be added in our implementation. When a job is submitted to Hadoop, JobTracker notifies MSJO that a job is added. MSJO puts the job into a queue. MSJO scheduler is event driven from JobTracker. When Hadoop is running, JobTracker keeps notifying MSJO on TaskTracker status. If a machine is idle, MSJO assigns a task to the TaskTracker of this machine. After a task is finished, the TaskTracker will tell JobTracker, which will further notify MSJO and MSJO updates job information. Accordingly, if all tasks in a job finish, MSJO removes the job and JobTracker sends a job-completion event to user application.

Here we also have a Processing Time Predictor module. This module can be based on a prediction algorithm or history recording of the completion time of past jobs. We leave such a prediction module to our future work. In this experiment, we run jobs one by one with default scheduler of Hadoop and collect data to train our predictor.

In current implementation, all algorithms are offline algorithms. They need full information about the job set before scheduling. To fulfill this requirement, we develop a job set generator. In each experiment, job set generator submits a set of jobs to Hadoop at the beginning of the experiment. These jobs carry release time and weight information with them. After MSJO collects all information for the job set, MSJO calls an algorithm module to make a schedule. After that, MSJO schedules jobs accordingly.

6.2. Experiment setup

We evaluate the algorithms with experiments on a 16-node cluster. This cluster is built on Amazon EC2. We choose virtual machines of type m1.small which have a 1-ECU cpu (1 ECU roughly equals to 1.0 GHz), 1.7 GB memory and 160 GB disk. According to our measurement, the inter-node network bandwidth is 400 Mbps.

We employ Wordcount as the MapReduce program in our experiments. Wordcount aims to count the frequency of words appearing in a data set. It is a benchmark MapReduce job and serves as a basic component of many Internet applications (e.g. document clustering, searching, etc.). In addition, many MapReduce jobs have aggregation statistics closer to Wordcount [9]. We use a document package from Wikipedia as input data of jobs. This package contains all English documents in Wikipedia since 30th January 2010 with uncompressed size of 43.7 GB. In this package, there are 27 individual files, of which the sizes range from 149 MB to 9.98 GB. For every file, we create a MapReduce job to process it. The number of map tasks is determined by input data size. One map task is created for 64 MB input data. We set the number of reduce tasks to half of the number of map tasks. The release time and job weights are generated in the same way as in the simulation.

We build two job sets: (1) Job set 1. It contains 10 jobs where input data size of every job is less than 1 GB. We use this job set to evaluate the performance of our algorithms when jobs are small. (2) Job set 2. This job set contains all 27 jobs.

6.3. Experiment result

Performance of different algorithms. For the job-level scenario, the result is shown in Fig. 18. In job set 1, we see that MarS outperforms the other algorithms. MarS increases 0.416 to the lower bound while MARES, HUWF and HJWF increase 0.539, 0.81 and 0.672 respectively. In job set 2, we see that MarS still outperforms rest of the algorithms. Compared with results in job set 1, we see TWJCT ratios of all algorithms increase. The trend is also reflected in simulation results. We notice that HJWF suffers more performance degradation than other algorithms. The reason may be that in job set 2, these jobs process data as large as 9.98 GB. Most of them are bigger than the jobs in job set 1. HJWF scheduled big jobs first because their weights are big. However, these jobs have big weights but their unit weights are small because they take a long time to process these data. As a result, small jobs with big unit weights are delayed. By considering the relation between weight and processing time, MarS, MARES and HUWF do not suffer from this mixture of different size jobs.

Furthermore, for the task-level scenario, as shown in Fig. 19, in both job sets 1 and 2, we observe that T-MarS outperforms the

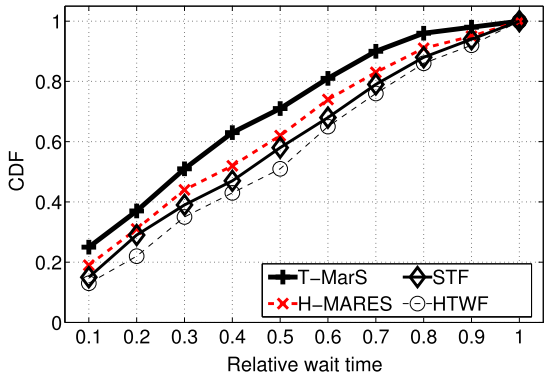

Fig. 16. Relative waiting time.

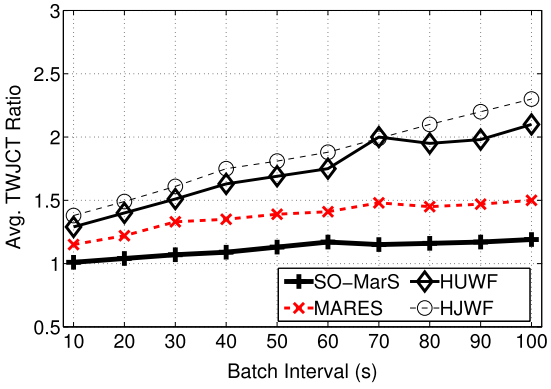

Fig. 17. Semi-Online MarS.

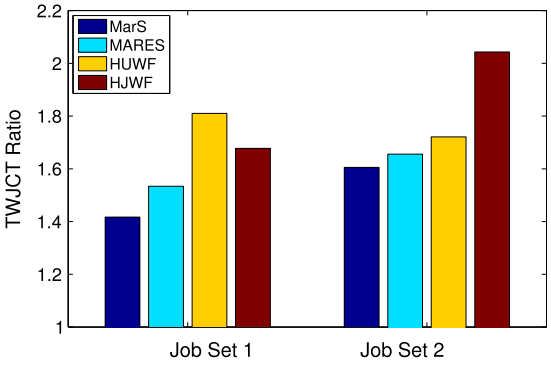

Fig. 18. TWJCT ratio in job level.

Fig. 19. TWTCT ratio in task level.

other algorithms. In particular, in job set 1, the difference of TWTCT ratio between T-MarS and H-MARES is small, and the performance

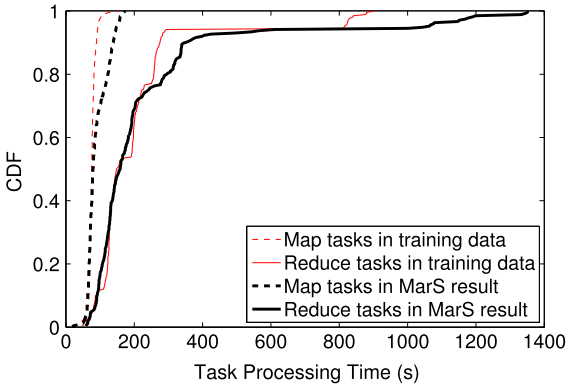

Fig. 20. Task processing time in experiment.

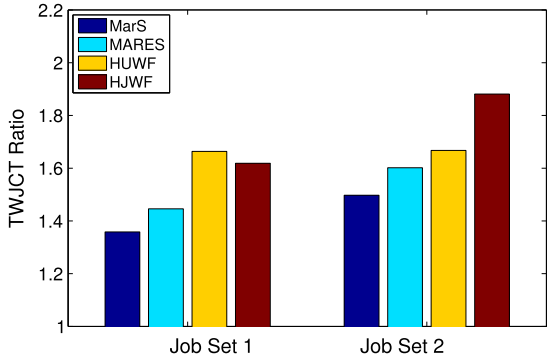

Fig. 21. Local server cluster experiment.

of STF is close to that of HTWF. In job set 2, we see that T-MarS still outperforms rest of the algorithms. Compared with results in job set 1, we see that all algorithms suffer performance degradation. However, the percentage of increasing TWTCT ratio of T-MarS is 3.9%, which is smaller than that of other algorithms. Moreover, compared to T-MarS, the difference of performance under job set 2 is larger than that under job set 1 for H-MARES, STF and HTWF.

Processing times of map tasks and reduce tasks. We show precessing times of map tasks and reduce tasks. Training data and MarS results are shown in Fig. 20. Both of them have 950 tasks. In training data, there is a clear processing time difference between reduce tasks and map tasks. Processing times of map tasks stay around 80 s while most of reduce task are over 150 s. We also see that there are gaps between training data and MarS result. The main reason is that training data is produced by running jobs in turn and the cluster is not fully utilized. Meanwhile, MarS schedules multiple jobs simultaneously to fully utilize the cluster. Intensive utilization of the cluster introduce cost from competitions on resources such as disk I/O, network bandwidth, etc.

6.4. Real server cluster experiment

To avoid the probable impact of virtual environment on the results, except for the above experiments on Amazon EC2, in this section we try to redo the experiment in a local real server cluster and perform comprehensive evaluations to demonstrate the superiority and reliability of our algorithm. The implementation is the same as Section 6.1, we evaluate the algorithms with experiments on a 12-server cluster. All the servers are connected with a 1 Gbps switch. Every server has 2 CPU cores, 80 GB RAM and 1 TB disk. We use the same workload and job sets as Section 6.2 in our experiments. As shown in Fig. 21, we can see that in job set 1, the TWJCT ratios of MarS, MARES, HUWF and HJWF are 1.36, 1.45, 1.68, 1.59 respectively; In job set 2, we see that MarS still outperforms rest of

the algorithms. Particularly, compared to MARES, the gain percentage of TWJCT ratio of MarS is 6.7%. This experiment indicates that our MarS scheduler still works extremely well when MapReduce framework runs on real server cluster.

7. Conclusion

In this paper, we studied MapReduce job scheduling with consideration of server assignment. We showed that without such joint consideration, there can be great performance loss. We formulated a MapReduce server-job organizer problem. This problem is NP-complete and we developed a 3-approximation algorithm MarS. Moreover, we further propose a novel fine-grained practical algorithm for general MapReduce task scheduling problem. Finally, we evaluated our algorithm through extensive simulation. The results show that MarS can outperform state-of-the-art strategies by as much as 40% in terms of total weighted job completion time. We also implement a prototype of MarS in Hadoop and test it with experiment on Amazon EC2. The experiment results confirm the advantage of our algorithm.

Acknowledgments

This work is supported by the National Basic Research Program of China under Grant No. 2012CB315806, the National Natural Science Foundation of China under Grant Nos. 61432009, 61170211, 61161140454, Specialized Research Fund for the Doctoral Program of Higher Education under Grant No. 20130002110058, and Joint Research Fund of MOE-China Mobile under Grant No. MCM20123041.

References

[1] Amazon EC2, 2012. http://aws.amazon.com/cn/ec2.
[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, E. Harris, Reining in the outliers in map-reduce clusters using Mantri, in: Proc. of USENIX OSDI, 2010.
[3] Apache Hadoop, 2012. http://hadoop.apache.org.
[4] Apache Mesos, 2015. http://mesos.apache.org.
[5] Apache YARN, 2015. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.
[6] E. Bortnikov, A. Frank, E. Hillel, S. Rao, Predicting execution bottlenecks in map-reduce clusters, in: Proc. of USENIX HotCloud, 2012.
[7] H. Chang, M. Kodialam, R.R. Kompella, T.V. Lakshman, M. Lee, S. Mukherjee, Scheduling in MapReduce-like systems for fast completion time, in: Proc. of IEEE INFOCOM, 2011.
[8] F. Chen, M. Kodialam, T.V. Lakshman, Joint scheduling of processing and shuffle phases in MapReduce systems, in: Proc. of IEEE INFOCOM, 2012.
[9] P. Costa, A. Donnelly, A. Rowstron, G. O'Shea, Camdoop: Exploiting in-network aggregation for big data applications, in: Proc. of USENIX NSDI, 2012.
[10] M.E. Crovella, M. Harchol-Balter, C.D. Murta, Task assignment in a distributed system: Improving performance by unbalancing load, in: Proc. of ACM SIGMETRICS, 1998.
[11] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: Proc. of USENIX OSDI, 2004.
[12] C. Delimitrou, C. Kozyrakis, Paragon: QoS-aware scheduling for heterogeneous datacenters, in: Proc. of ACM ASPLOS, 2013.
[13] C. Delimitrou, C. Kozyrakis, Quasar: Resource-efficient and QoS-aware cluster management, in: Proc. of ACM ASPLOS, 2014.
[14] D. DeWitt, M. Stonebraker, MapReduce: A major step backwards, 2008. http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html.
[15] M.R. Garey, D.S. Johnson, Computers and Intractability; A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., 1990.
[16] R.L. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Appl. Math. 17 (2) (1969) 416–429.
[17] M. Harchol-Balter, Task assignment with unknown duration, J. ACM 49 (2) (2002) 260–288.
[18] C. He, Y. Lu, D. Swanson, Matchmaking: A new MapReduce scheduling technique, in: Proc. of IEEE CloudCom, 2011.
[19] C. He, Y. Lu, D. Swanson, Real-time scheduling in MapReduce clusters, in: Proc. of IEEE HPCC and EUC, 2013.
[20] H. Herodotou, F. Dong, S. Babu, No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics, in: Proc. of ACM SoCC, 2011.
[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: Fair scheduling for distributed computing clusters, in: Proc. of ACM SOSP, 2009.
[22] B.W. Lampson, A scheduling philosophy for multiprocessing systems, Commun. ACM 11 (5) (1968) 347–360.
[23] S. Li, S. Hu, T. Abdelzaher, The packing server for real-time scheduling of MapReduce workflows, in: Proc. of IEEE RTAS, 2015.
[24] S. Li, S. Hu, S. Wang, L. Su, T. Abdelzaher, I. Gupta, R. Pace, WOHA: Deadline-aware Map-Reduce workflow scheduling framework over hadoop clusters, in: Proc. of IEEE ICDCS, 2014.
[25] B. Palanisamy, A. Singh, L. Liu, B. Jain, Purlieus: Locality-aware resource allocation for mapreduce in a cloud, in: Proc. of ACM SC, 2011.
[26] M. Queyranne, Structure of a simple scheduling polyhedron, Math. Program. 58 (2) (1993) 263–285.
[27] M. Queyranne, A. Schulz, Approximation bounds for a general class of precedence constrained parallel machine scheduling problems, SIAM J. Comput. 35 (5) (2006) 1241–1253.
[28] T. Sandholm, K. Lai, Mapreduce optimization using regulated dynamic prioritization, in: Proc. of ACM SIGMETRICS, 2009.
[29] A.S. Schulz, et al. Polytopes and scheduling, Technical University of Berlin, 1996.
[30] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, J. Wilkes, Omega: Flexible, scalable schedulers for large compute clusters, in: Proc. of ACM EuroSys, 2013.
[31] J. Tan, S. Meng, X. Meng, L. Zhang, Improving reducetask data locality for sequential mapreduce jobs, in: Proc. of IEEE INFOCOM, 2013.
[32] A. Verma, L. Cherkasova, R.H. Campbell, ARIA: Automatic resource inference and allocation for mapreduce environments, in: Proc. of ACM ICAC, 2011.
[33] B. Wang, J. Jiang, G. Yang, ActCap: Accelerating mapreduce on heterogeneous clusters with capability-aware data placement, in: Proc. of IEEE INFOCOM, 2015.
[34] W. Wang, K. Zhu, L. Ying, J. Tan, L. Zhang, Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality, in: Proc. of IEEE INFOCOM, 2013.
[35] D. Xie, N. Ding, Y.C. Hu, R. Kompella, The only constant is change: Incorporating time-varying network reservations in data centers, in: Proc. of ACM SIGCOMM, 2012.
[36] Y. Yuan, H. Wang, D. Wang, J. Liu, On interference-aware provisioning for cloud-based big data processing, in: Proc. of IEEE/ACM IWQoS, 2013.
[37] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, I. Stoica, Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling, in: Proc. of ACM EuroSys, 2010.
[38] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, Improving MapReduce performance in heterogeneous environments, in: Proc. of USENIX OSDI, 2008.
[39] W. Zhang, S. Rajasekaran, T. Wood, M. Zhu, MIMP: Deadline and interference aware scheduling of hadoop virtual machines, in: Proc. of IEEE/ACM CCGrid, 2014.
[40] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J.Y. Li, W. Lin, J. Zhou, L. Zhou, Optimizing data shuffling in data-parallel computation by understanding user-defined functions, in: Proc. of USENIX NSDI, 2012.
[41] Y. Zheng, N. Shroff, P. Sinha, A new analytical technique for designing provably efficient mapreduce schedulers, in: Proc. of IEEE INFOCOM, 2013.

Xiao Ling received the B.Sc. degree from Beijing University of Posts and Telecommunications. He is now a Ph.D. candidate at the Department of Computer Science and Technology in Tsinghua University. He was a visiting Ph.D. student at the Hong Kong Polytechnic University between 2014 and 2015. His major research interests include cloud computing, distributed system and big data processing applications. He is a student member of IEEE.

Yi Yuan received the B.Sc. degree and the M.Sc. degree from University of Electronic Science and Technology of China, Chengdu, China. He received the Ph.D. degree from The Hong Kong Polytechnic University, Hong Kong. He is currently a technical engineer in Cloud Computing Department, Tencent Company. His major research interests include cloud computing, distributed system and green building. He is a student member of IEEE.

Dan Wang received the B.Sc. degree from Peking University, Beijing, China, the M.Sc. degree from Case Western Reserve University, Cleveland, Ohio, USA, and the Ph.D. degree from Simon Fraser University, Burnaby, B.C., Canada; all in computer science. He is an Assistant Professor of Department of Computing, The Hong Kong Polytechnic University, Hong Kong. His research interests include wireless sensor networks, Internet routing and cloud computing. He is a senior member of IEEE.

Jiangchuan Liu received the B.Sc. degree from Tsinghua University, Beijing, China, and the Ph.D. degree from The Hong Kong University of Science and Technology, Hong Kong. From 2003 to 2004, he was an Assistant Professor in the Department of Computer Science and Engineering at The Chinese University of Hong Kong. He was a Microsoft Research Fellow, and worked at Microsoft Research Asia (MSRA) in the summers of 2000, 2001, 2002, 2007, and 2011. He is a Full Professor in the School of Computing Science at Simon Fraser University, Canada, and an EMC-Endowed Visiting Chair Professor of Tsinghua University, Beijing, China. His research interests include multimedia communications, peer-to-peer networking, cloud computing, online gaming, social networking, big data networking, and wireless sensor/mesh networking. He is a senior member of IEEE.

Jiahai Yang received the B.Sc. degree from Beijing Technology and Business University, the M.Sc. degree and Ph.D. degree from Tsinghua University, Beijing, China; all in Computer Science. He is a Professor of the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China. His research interests include network management, network measurement, Internet routing and applications, cloud computing and big data applications. He is a member of IEEE & ACM.