Joint Scheduling of MapReduce Jobs with Servers: Performance Bounds and Experiments

Yi Yuan*, Dan Wang*†, Jiangchuan Liu‡
*Department of Computing, The Hong Kong Polytechnic University
†The Hong Kong Polytechnic University Shenzhen Research Institute
‡School of Computing Science, Simon Fraser University
Email: {csyiyuan, csdwang}@comp.polyu.edu.hk, jcliu@cs.sfu.ca

***Abstract*—MapReduce has 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. We evaluate our algorithms through both simulations and experiments on Amazon EC2 with an implementation in Hadoop. The results confirm the advantage of our algorithms.**

I. 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 [1], 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" [2]. 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] yet reduce tasks must wait until all map tasks in the same job finish. 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.

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

978-1-4799-3360-0/14/$31.00 ©2014 IEEE

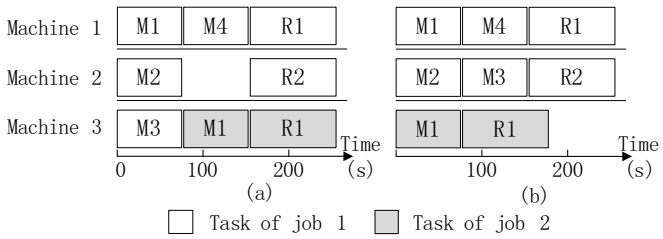

Fig. 1: Impact of server assignment. (a) Without server assignment, Hadoop default strategy. (b) Joint considering of server assignment.

In this paper, we conduct research in this direction. There are recent studies on MapReduce scheduling [3][4]. As an example, an algorithm is developed in [4] for joint scheduling of processing and shuffle phases and it achieves an 8-approximation. All past studies 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 the impact that if the server assignment is "less good", whether this will affect the scheduling on map and reduce tasks. 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 task. Job 2 has 1 map task and 1 reduce task. Assume the processing time to be 75 seconds for all map tasks and 100 seconds for all reduce tasks. If server assignment is not considered, we will result in Fig.1(a), which follows the default FIFO strategy of Hadoop [5]. However, if we joint 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 seconds and in Fig.1(b) is 175 seconds, 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[6]. The experiment results confirm the advantage of our algorithm.

The remaining part of the paper is organized as follows. We discuss related work in Section II. We formulate the MJSO problem and analyze its complexity in Section III. In Section IV, we present a set of algorithms. We evaluate our algorithms in Section V. In Section VI, we show an implementation of our scheme in Hadoop and our evaluation in Amazon EC2. Finally, Section VII concludes the paper.

II. Related Work

Due to the widely usage of MapReduce systems, there is a flourish of studies on understanding MapReduce performance and many developed various improvement schemes. From system point of view, there are many valuable advances on improving data-shuffling [7], in-network aggregation [8], etc. From algorithmic point of view, people are looking into MapReduce job/task scheduling with various considerations in different scenarios. Quincy [9] and delay scheduling [10] are proposed to provide fair scheduling for MapReduce systems. Omega [11] is proposed to support cooperation of multiple schedulers in large computer clusters. Zheng et al.[12] propose a MapReduce scheduler with provable efficiency on total flow time. Data locality is considered where Wang et al. [13] investigate map task scheduling under heavy traffic and Tan et al. [14] propose a stochastic optimization framework to optimize scheduling of reduce tasks.

Two most closely related works of our paper are [3][4]. In [3], Chang et al. propose scheduling algorithms for fast completion time. In [4], Chen et al. investigate precedence constraints between map tasks and reduce tasks in MapReduce job, and propose a 8-approximation algorithm. 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 study. There are many works with inspiring ideas and analytical techniques [15][16]. For minimizing total weighted job completion time with precedence constraints, the best known work is a 4-approximation algorithm in [17]. However, these works focus on the general case.

III. MapReduce Server-Job Organizer: The Problem and Complexity Analysis

A. 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 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. 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 phrase). 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 constraints for every job j.

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

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

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

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

B. 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 shown MSJO is NP-complete, we reduce a job schedule problem (problem SS13 in [18]) to it. Problem SS13 is proven NP-complete. The proven theorem can be stated as follow: *Given a set $\mathbb{J}$ of jobs and a set $\mathbb{M}$ of identical machines, every job j has a weight w_j. Every job j can be processed uninterruptedly on every machine with processing time p_j. Let C_j be job completion time of job j in a feasible schedule. It is NP-complete to determine a feasible schedule where $\sum_{j\in\mathbb{J}} w_j C_j$ is minimized.*

Given every instance $(\mathbb{J}, \mathbb{M})$ of problem SS13, 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. ■

IV. 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 is

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 factor of the optimal solution for MSJO.

A. 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 follow:

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

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 Equation.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 Equation.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 Equation.4 increases and the relation still holds.

B. Conditional LP Relaxation and Constraint Generation Algorithm

Note that there are an exponential number of constraints in Equation.4 due to the exponential number of $\mathbb{B}$. In theory, CLS-LPP can be solved in polynomial time using ellipsoid method with a separation oracle [16]. However, the complexity associated with ellipsoid method makes it impractical to solve large problems with thousands of decision variables. We derive a new LP-relaxation problem which has a small subset of constraints in Equation.4. The optimal result of this new LP-relaxation problem also leads to the 3-approximation algorithm to be developed in section IV-C. 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).

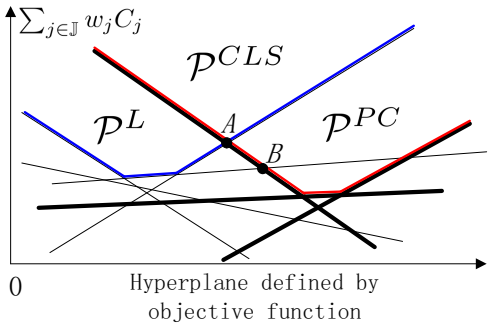

Fig. 2: Illustration of optimal solutions of CLS-LPP and CND-LPP. Fine black lines represent constraints in Equation.4. Thick black lines represent constraints in Equation.1-3. Blue lines represent boundary of $\mathcal{P}^L$. Red lines represent boundary of $\mathcal{P}^{PC}$.

Before developing our algorithm for building CND-LPP, we introduce a property of the solutions that satisfy Equation.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 Equation.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 $i \in [2\ldots|\mathbb{T}|]$:

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

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 Equation.4 as:

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

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 Equation.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 Equation.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 Equation.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 Equation.1-3 define another polyhedrion, denoted as $\mathcal{P}^{PC}$. We know that $\mathcal{P}^{CLS} = \mathcal{P}^L \cap \mathcal{P}^{PC}$ (see Fig.2). 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.2), we search a solution in $\mathcal{P}^{PC}$ (point B in Fig.2) which has Property.1. We start with an initial CND-LPP which only contains all precedence constraints in Equation.1-3. By iteratively adding fine chosen constraints in Equation.4, we approach the desirable solution.

To check whether an optimal solution satisfies Property.1, we can check whether constraints in Equation.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 $\forall i \in [2 \dots |\mathbb{T}|]$, 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 Equation.1-3 are included. Its objective function is same to 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 $\forall i \in [1 \dots |\mathbb{T}|]$. If $\mathbb{S}^{LP}$ satisfies all PGCs, we are done. Otherwise, we add the violated PGCs to CND-LPP and repeat step 1 to 3 until we produce an optimal solution 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 a offset which idicates how much $S_{\pi(i)}$ should increase to satisfy $PGC(\pi, i)$. Function $ViolationOffset(PGC(\pi, i), \mathbb{S}^{LP})$ calculate 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 $k \in [i-1 \dots |\mathbb{T}|]$. Finally, after all PGCs are checked, we check whether $(\mathbb{S}^{LP}, \mathbb{C}^{LP})$ reaches the stop threshold.

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 Equation.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^LP_j ≥ (1 − ε) Σ_{j∈J} w_j C^NV_j then
            return (S^LP, C^LP);
until vn is 0
return (S^LP, C^LP);
```

Algorithm 2 MarS()

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

```
S^H ← ∅;
π ← Sort tasks by middle finish time according to S^LP;
Let e_m be earliest idle time of machines m;
e_m ← 0, ∀m ∈ M;
for all i ∈ [1 ... |T|] do
    Find job j where π(i) ∈ T_j^(M) or π(i) ∈ T_j^(R);
    if π(i) ∈ T_j^(M) then
        S^earlist = r_j;
    if π(i) ∈ T_j^R then
        S^earlist = max_{u∈T_j^(M), S_u^H ∈ S^H} {S_u^H + p_u + d_{uπ(i)}};
    Find m* where e_m* = min_{m∈M} {e_m};
    S_v^H = max{S^earlist, e_m*};
    e_m* = S^H_π(i) + p_π(i);
    S^H ← S^H ⋃ S^H_π(i);
return S^H;
```

C. MarS Algorithm

In this section, we describe MarS. MarS is an 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 $\min\{S^{earliest}, e_{m^*}\}$ 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 $\pi(k)$, $k \in [1 \dots (i-1)]$, are scheduled by MarS, if we choose any machine m and start $\pi(i)$ as soon as possible after machine m is idle, it result 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:

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

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 task u_2 and v_3 in Fig.3 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 $S^{LP}_{u_2} \geq S^{LP}_{u_1} + d_{u_1u_2} + p_{u_1}$. Our idea is to prove $S^H_{u_2} - (S^H_{v_3} + p_{v_3}) \leq M^{LP}_{u_2} - M^{LP}_{v_3}$ 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 side of these inequations, we have Equation.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 task 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^I to represent the start point of the schedule. Then, constraints $S_u \geq r_j, \forall u \in \mathbb{T}^{(M)}_j$ in Equation.1 can be expressed in the form of precedence constraint with delay: $S_u \geq S_{t^I} + d_{t^Iu} + p_{t^I}, \forall u \in \mathbb{T}^{(M)}_j$ where $S_{t^I} = 0, d_{t^Iu} = r_j, p_{t^I} = 0$. 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* $\{u_1 \to u_2 \to \dots \to u_s\}$ where link $(u_k, u_{k+1}) \in L$ is tight for all $k \in [1 \dots s-1], s \geq 2$. Fig.3 demonstrates a three-node tight-link path in a schedule result. If $u_s = u$, we call $\{u_1 \to u_2 \to \dots \to u_s\}$ *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 $u \in T^{(M)}_j$, $LTLP(u)$ can be empty or

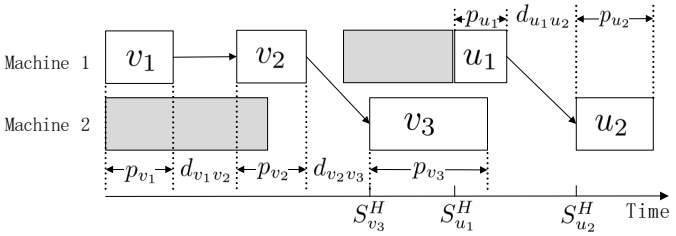

Fig. 3: Demo of tight-link path $v_1 \to v_2 \to v_3$ and $u_1 \to u_2$ in a schedule result for proof of Lemma 2.

$LTLP1 = \{t^I \to u\}$. For a reduce task $v \in T^{(R)}_j$, $LTLP(v)$ can be $LTLP2 = \{t^I \to u \to v\}$ or $LTLP3 = \{u \to v\}$ where $u \in T^{(M)}_j$.

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

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

Then we have:

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

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

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

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

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

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 Equation.8 and Equation.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 $LTLP(\hat{u}) = \{u_1 \to u_2 \to \dots \to u_s\}$. Because u_1 is the first task in $LTLP(u_s)$, there must be a task $\hat{v}$ scheduled on machine m where $S_{\hat{v}} \geq S_{u_1} \geq (S_{\hat{v}} + p_{\hat{v}})$ (see task v_3 in Fig.3) and $M^{LP}_{\hat{v}} \geq M^{LP}_{u_1}$. Otherwise, u_1 can start earlier. With Equation.8 and Equation.9, we have:

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

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^I whose middle finish time is 0. By adding all these inequations together, we have Equation.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.

Theorem 3: MarS is a 3-approximation algorithm for M-SJO.

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 Equation.5 and Equation.7, we have:

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

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

$$S^{H}_{\pi(i)} \leq \sum_{u \in \mathbb{U}(m,\pi(i))} p_u + g(m, \pi(i)), \forall m \in \mathbb{M}$$

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

$$S^{H}_{\pi(i)} + p_{\pi(i)} \leq 3M^{LP}_{\pi(i)} + \frac{3}{2}p_{\pi(i)} = 3(S^{LP}_{\pi(i)} + p_{\pi(i)}) \quad (10)$$

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

$$\sum_{j \in \mathbb{J}} w_j C^{H}_{j} \leq 3\sum_{j \in \mathbb{J}} w_j C^{LP}_{j}$$

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

V. Simulation

A. Simulation Setup

1) Background: We use synthetic workloads to study the performance of our algorithm, following similar simulation set-up in [3][4]. 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 follow 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 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.

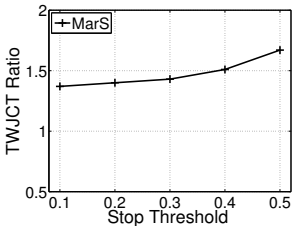

Fig. 4: Stop threshold vs TWJCT ratio.

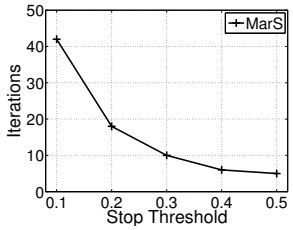

Fig. 5: Stop threshold vs iteration number.

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

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

H-MARES: H-MARES [4] is a LP-based heuristic algorithm considering precedence constraints in MapReduce jobs. In evaluation of [4], H-MARES outperforms other algorithms with a factor of 1.5 to the lower bound. 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. 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 [16], when there is not release time and precedence constraints, LP relaxation constraints in Equation.4 and constraints in [4] 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 a 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.

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 MarS, we need to choose a stop threshold ϵ for COGE. We run MarS with 100 jobs and change value of stop thresholds (see Fig.4 and Fig.5). We see that when ϵ changes from 0.1 to 0.5, there is a small performance degradation for MarS

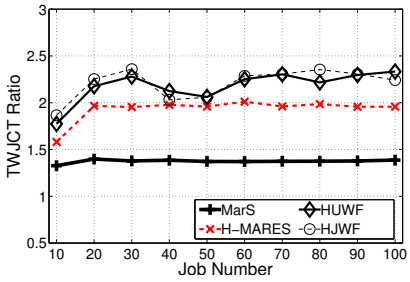

Fig. 6: Performance comparison for most randomized scenario. All job parameters are in random categories. Machine number is 50.

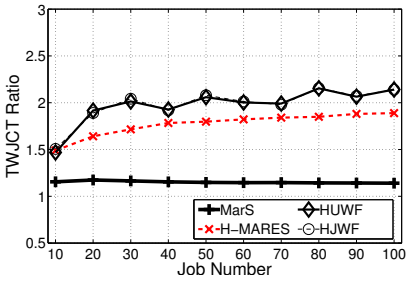

Fig. 7: Performance comparison for uniform task number scenario. Other job parameters are in random categories. Machine number is 50.

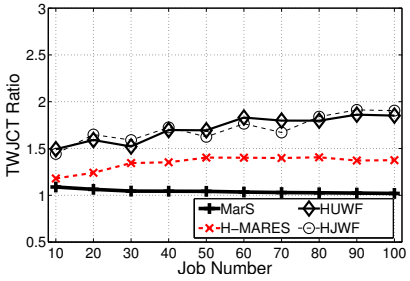

Fig. 8: Performance comparison for uniform task processing time scenario. Other job parameters are in random categories. Machine number is 50.

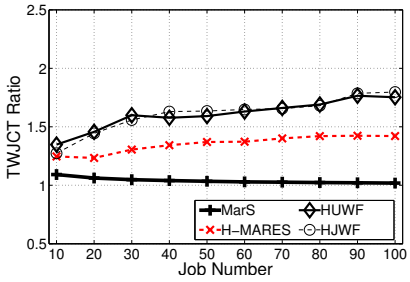

Fig. 9: Performance comparison for most randomized scenario. All job parameters are in random categories. Machine number is 100.

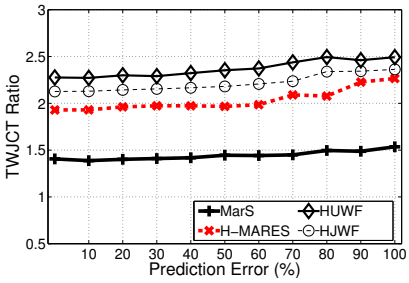

Fig. 10: Impact of prediction error for most randomized scenario. Job number is 100. Machine number is 50.

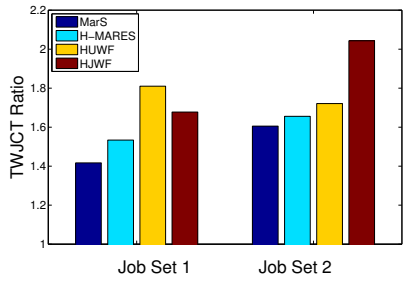

Fig. 11: Experiment results for different job set sizes.

while iteration number decreases rapidly. Thus, we choose $\epsilon = 0.3$. H-MARES also need a stop threshold in solving its LP-relaxation problem. To offer a fair comparison, we choose $\epsilon = 0.3$ for H-MARES instead of 0.5 in the original paper [4].

In our simulation, we assume that the task processing time and delays between a map task and a reduce task are known to the scheduler. There are many studies on accurate processing time prediction [19][20] and trace study [21] shows that the majority of map tasks and reduce tasks are highly recurrent making prediction feasible. We plan a future work here.

B. 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.

Performances in the most randomized scenario. Fig.6 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. H-MARES shows stable performance after job number reaches 20. However, we see a constant performance difference between H-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 H-MARES is 1.58, HUWF is 1.77 and HJWF is 1.86. After job number rises to 20, H-MARES increases to 1.96, HUWF and HJWF jump to 2.17 and 2.25 respectively. This is because when there is 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 H-MARES outperforms HUWF and HJWF in all cases. This is because workload-based allocation performs well and H-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.7. Compared with results in Fig. 6, we see that all algorithms gain better performance. MarS stays at about 1.15 which is 0.17 lower in terms of TWJCT ratio. H-MARES gradually increase 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 H-MARES is reduced to 0.35 on average.

It is worth noticing that comparing results in Fig.7 and Fig.8, 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.9). We see that MarS performs extremely well. Starting at 1.09 when job number is 10, its TWJCT ratio gradually decline 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.6). Different from MarS, TWJCT ratios of other algorithms increase with job number. When job number reaches 100, MarS outperform H-MARES, HUWF and HJWF by 0.42, 0.75 and 0.79 respectively.

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.10. x-axis is maximum prediction error to no-error precessing time. We see that all algorithms do not show great performance degradation. TWJCT ratios of four algorithms increase slightly. The maximum performance degradation is found in H-MARES but it is less than 0.35 in term of TWJCT ratio when maximum prediction error is 100%.

VI. Implementation and Results of Experiment

A. Implementation

We implement a MSJO framework in Hadoop-1.2.0 and run the implementation on Amazon EC2. The implementation framework is described in Fig.12. 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 module. Currently, we implemented four algorithm modules for MarS, H-MARES, HUWF and HJWF in our experiments. Other algorithm modules can be added, and we open source our implementation at [22]. 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.

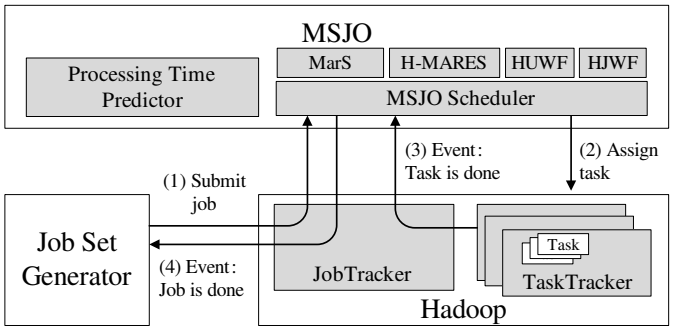

Fig. 12: Processing of a job in MarS implement with Hadoop.

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 call an algorithm module to make a schedule. After that, MSJO schedules jobs accordingly.

B. 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 160GB disk. According to our measurement, the inter-node network bandwidth is 400Mbps.

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 [8]. We use a document package from Wikipedia as input data of jobs. This package contains all English documents in Wikipedia since 30^{th} 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 1GB. 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.

C. Experiment Result

Performance of different algorithms The result is shown in Fig.11. In job set 1, we see that MarS outperforms the

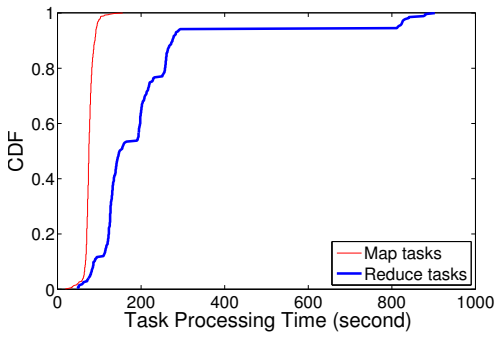

Fig. 13: Comparison of task processing time in experiment.

other algorithms. MarS increases 0.416 to the lower bound while H-MARES, HUWF and HJWF increase 0.539, 0.81 and 0.672 respectively. In job set 2, we see that MarS still outperforms rest 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. They are some relatively big to 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, H-MARES and HUWF do not suffer from this mixture of different size jobs.

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.13. 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 stays around 80 seconds while most of reduce task are over 150 seconds. 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.

VII. 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. 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.

Acknowledgement

Dan Wang's work is supported by National Natural Science Foundation of China (No. 61272464), RGC/GRF PolyU 5264/13E, HK PolyU 1-ZVC2, G-UB72. Jiangchuan Liu's work is supported by a Canada NSERC Discovery Grant, an NSERC Strategic Project Grant, and a China NSFC Major Program of International Cooperation Grant (61120106008).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[2] D. DeWitt and M. Stonebraker, "Mapreduce: A major step backwards," 2008. http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html.

[3] H. Chang, M. Kodialam, R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee, "Scheduling in mapreduce-like systems for fast completion time," in *Proc. IEEE INFOCOM*, 2011.

[4] F. Chen, M. Kodialam, and T. Lakshman, "Joint scheduling of processing and shuffle phases in mapreduce systems," in *Proc. IEEE INFOCOM*, 2012.

[5] "Apache hadoop," 2013. http://hadoop.apache.org/.

[6] "Amazon ec2," 2013. http://aws.amazon.com/cn/ec2/.

[7] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou, "Optimizing data shuffling in data-parallel computation by understanding user-defined functions," in *Proc. USENIX NSDI*, 2012.

[8] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: exploiting in-network aggregation for big data applications," in *Proc. USENIX NSDI*, 2012.

[9] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proc. ACM SOSP*, 2009.

[10] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proc. ACM EuroSys*, 2010.

[11] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proc. ACM EuroSys*, 2013.

[12] Y. Zheng, N. B. Shroff, and P. Sinha, "A new analytical technique for designing provably efficient mapreduce schedulers," in *Proc. IEEE INFOCOM*, 2013.

[13] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang., "Map task scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," in *Proc. IEEE INFOCOM*, 2013.

[14] J. Tan, S. Meng, X. Meng, and L. Zhang., "Improving reducetask data locality for sequential mapreduce jobs," in *Proc. IEEE INFOCOM*, 2013.

[15] B. W. Lampson, "A scheduling philosophy for multiprocessing systems," *Commun. ACM*, vol. 11, pp. 347–360, May 1968.

[16] A. S. Schulz *et al.*, *Polytopes and scheduling*. PhD thesis, Technical University of Berlin, 1996.

[17] M. Queyranne and A. S. Schulz, "Approximation bounds for a general class of precedence constrained parallel machine scheduling problems," *SIAM Journal on Computing*, vol. 35, no. 5, pp. 1241–1253, 2006.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[19] A. Verma, L. Cherkasova, and R. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proc. ACM ICAC*, 2011.

[20] Y. Yuan, H. Wang, D. Wang, and J. Liu, "On interference-aware provisioning for cloud-based big data processing," in *Proc. IEEE/ACM IWQoS*, 2013.

[21] E. Bortnikov, A. Frank, E. Hillel, and S. Rao, "Predicting execution bottlenecks in map-reduce clusters," in *Proc.USENIX HotCloud*, 2012.

[22] Y. Yuan, D. Wang, and J. Liu, "Joint scheduling of mapreduce jobs with servers: Performance bounds and experiments (msjo package)," 2014. http://www4.comp.polyu.edu.hk/~csyiyuan/projects/MarS/MarS.html.