Joint Scheduling of Processing and Shuffle Phases in MapReduce Systems

Fangfei Chen*, Murali Kodialam†, T. V. Lakshman†
*Department of Computer Science and Engineering, The Penn State University
† Bell Laboratories, Alcatel-Lucent

Abstract—**MapReduce has emerged as an important paradigm for processing data in large data centers. MapReduce is a three phase algorithm comprising of *Map*, *Shuffle* and *Reduce* phases. Due to its widespread deployment, there have been several recent papers outlining practical schemes to improve the performance of MapReduce systems. All these efforts focus on one of the three phases to obtain performance improvement. In this paper, we consider the problem of jointly scheduling all three phases of the MapReduce process with a view of understanding the theoretical complexity of the joint scheduling and working towards practical heuristics for scheduling the tasks. We give guaranteed approximation algorithms and outline several heuristics to solve the joint scheduling problem.**

I. Introduction

MapReduce [1] has emerged as a significant data processing paradigm in data centers. MapReduce is used in several applications including searching the web, URL frequency estimation and indexing. In a typical application, the data on which MapReduce operates is partitioned into chunks and assigned to different processors. MapReduce is essentially a three step process:

1) In the *Map* phase, several parallel tasks are created to operate on the relevant data chunks to generate intermediate results. These results are stored in the form of key–value pairs.
2) In the *Shuffle* phase, the partial computation results of the map phase are transferred to the processors performing the reduce operation.
3) During the *Reduce* phase, each processor that executes the reduce task aggregates the tuples generated in the map phase.

A frequent requirement for routine data center requests is fast response time. In a large data center where there are several MapReduce jobs that run concurrently, a centralized *master* coordinates the assignment and scheduling of MapReduce tasks across the data center. The assignment problem is to decide which processor will execute a map or reduce task and the scheduling is to decide in what order the tasks will be executed on each processor. In this paper, we assume that tasks are already assigned to the processors and the focus of the paper is to determine the schedule that minimizes the mean response time over all jobs. MapReduce throws up several significant scheduling challenges due to the dependencies between different phases of a job's MapReduce operations as well as dependencies between different processors that is the consequence of splitting a job across multiple processors. Though dependencies between tasks have been studied in the job shop scheduling literature, MapReduce system presents several new challenges. For example, in traditional job shops, each job comprises of multiple tasks but at most one task can be executed at any given time unlike MapReduce scheduling where tasks may be executed in parallel. The shuffle operation in MapReduce assumes simultaneous possession of multiple resources (the egress of the transmit processor and the ingress of the receive processor) and this phase resembles data migration problems more than job-shop problems.

If the shuffle operation is not a bottleneck, there is no advantage if one map task belonging to a particular job is completed early while another task belonging to the same job is delayed. This is the key idea in the MapReduce formulation in Chang *et al.* [2]. Recent studies by Chowdhury *et al.* [3] have shown that the shuffle operation accounts for as much as a third of the completion time of a MapReduce job. Therefore the shuffle operation has to be taken into consideration explicitly in the scheduling problem. Including the shuffle operation introduces new tradeoffs in the scheduling problem. If all the map tasks belonging to a job complete almost simultaneously, and all the shuffle operations for this job can start at this time. This creates a bottleneck at the shuffle operation. Therefore, once shuffle is included in the system, it may be better to spread the completion of the map task in order to smoothly schedule the shuffle operation.

The widespread use of MapReduce has lead to several recent papers [4], [5], [6] outlining practical approaches to improving the performance of MapReduce systems. Though these approaches are shown to improve the performance of specific parts of MapReduce, they do not address the end-to-end performance of MapReduce. Unlike the formulation in Chang *et al.* [2], we explicitly model the precedence between the map and reduce operations in this paper and outline a constant factor approximation algorithm. Moreover, [2] ignores the shuffle phase. The approach in this paper is to develop a theoretical framework to model the end-to-end performance of MapReduce systems and in particular model the interactions between the three phases of MapReduce. To our knowledge, this is the first paper that addresses map-shuffle-reduce from a theoretical viewpoint.

This paper makes the following contributions:

1) We propose a formulation for explicitly taking into consideration dependencies between map and reduce operations and outline a constant factor guaranteed ap-

proximation algorithm (MARES) for the problem.

2) The approximation algorithm involves solving a linear programming problem with exponential number of constraints and we outline a column-generation based approach to solve the linear programming relaxation.
3) We develop a constant factor approximation algorithm for the map-shuffle-reduce problem (MASHERS).
4) We develop heuristics for both the map-reduce (H-MARES) as well as map-shuffle-reduce (D-MASHERS, H-MASHERS) based on the constant factor approximation algorithms and study their performance. We show via experiments that these heuristics perform extremely well in practice.

The rest of the paper is organized as follows. Section II formally defines the scheduling problem with precedence constraints, provides a LP-based lower bound and shows how to solve the LP. Section III presents the approximation algorithm for the two phase map-reduce problem. Section IV introduce the shuffle phase into the problem and presents the approximation algorithm for the map-shuffle-reduce problem. Section V evaluates these heuristics by simulations. Section VI discusses several related work and finally Section VII concludes the paper.

II. Scheduling Dependent Tasks: A Column Generation Approach

In this section we consider the problem of scheduling a set of dependent tasks on multiple processors in order to minimize the sum of the weighted completion time of the tasks. We develop a linear programming based approach to get a lower bound on the optimal solution. Solution to this linear programming problem exploits the structure of the single processor scheduling polyhedron. The linear programming solution is infeasible to the scheduling problem but we exploit the structure of the dependencies in MapReduce to derive approximation algorithms for both the two phase map-reduce problem and the three phase map-shuffle-reduce problem.

A. Problem Definition

We are given a set J of jobs and a set P of processors. Job $j \in J$ has a set of Tasks T_j. Let $T = \cup_j T_j$ represent the set of all tasks. Each task u is assigned to a processor $p(u) \in P$ and its processing time $t_u \geq 0$. The set of tasks assigned to processor p is denoted by J_p. Let n_p denote the number of tasks assigned to processor p. We assume that a processor can execute at most one task at any given time and all tasks are completed in a non-preemptive manner. Let $G = (V,\ L)$ denote the *Precedence Graph* among tasks, where the set of nodes is the set of tasks and $(u, v) \in L$ indicates that task v can only start after the completion of task u. Let C_u denote the completion time of task u. Associated with task u is a weight w_u that indicates its relative importance and and the overall scheduling objective is to minimize the weighted sum of the task completion times $\sum_u w_u C_u$.

The problem of scheduling tasks with precedence constraints in order to minimize the total weighted completion time is NP-hard [7] even on a single processor. Our problem is a significant generalization of this problem and is therefore NP-hard.

B. LP Based Lower Bound

Consider a processor p and the set of tasks J_p assigned to this processor. The scheduling problem on processor p is to decide the order in which the tasks in J_p are processed. It can be shown (see [13] and the references in it) that the completion time C_u for the tasks $u \in J_p$ lies in the following polyhedron denoted by $\mathcal{P}_p$:

$$\sum_{u \in S} t_u C_u \geq f(S, p) \ \ \forall S \subseteq J_p,\ \forall p \tag{1}$$

where

$$f(S,p) = \frac{1}{2}\left[\sum_{u \in S} t_v^2 + \left(\sum_{u \in S} t_v\right)^2\right]$$

The intuition behind these inequalities is simple. Consider a processor on which two tasks with processing times t_1 and t_2 have to be processed. If the tasks are processed in the order $1 \rightarrow 2$, then the completion times of the two tasks will be $C_1 = t_1$ and $C_2 = t_1 + t_2$. If the order is reversed then $C_1 = t_1 + t_2$ and $C_2 = t_2$. Note that $t_1C_1 + t_2C_2 = t_1^2 + t_2^2 + t_1t_2 = \frac{1}{2}(t_1^2 + t_2^2 + (t_1 + t_2)^2)$ in either of the cases. This argument can be extended to any subset of jobs. Note that $\mathcal{P}_p$ represents only the set of necessary conditions that the completion times C_u have to satisfy. The polyhedron has many vectors that are not achievable by any scheduling policy. It can be shown that $\mathcal{P}_p$ has $n_p!$ extreme points, each corresponding a permutation of the tasks in J_p. A given permutation of tasks in J_p, results in a set of completion times of the tasks in J_p. This vector of completion times represents an extreme point of the polyhedron $\mathcal{P}_p$. Let E_p^k represent extreme point k of the polyhedron $\mathcal{P}_p$. Note that E_p^k is an n_p dimensional vector. We use $E_p^k(u)$ to denote the completion time of job $u \in J_p$ corresponding to extreme point k at processor p. An alternate way to represent $\mathcal{P}_p$ is to write it as a convex combination of its extreme points. We use λ_p^k to represent the non-negative weight associated with extreme point k in polyhedron for processor p.

$$\mathcal{P}_p = \left\{ \sum_{k=1}^{n_p!} \lambda_p^k E_p^k : \sum_{k=1}^{n_p!} \lambda_p^k = 1\ ,\ \lambda_p^k \geq 0 \right\}$$

Since the polyhedron $\mathcal{P}_p$ represents the set of necessary conditions for the completion time of the tasks on processor p, the linear programming problem:

$$\begin{aligned} &\min \sum_{u \in T} w_u C_u \\ &C_u \in \mathcal{P}_{p(u)} \\ &C_v \geq C_u + t_v \qquad\qquad \forall (u,v) \in L \end{aligned}$$

gives a lower bound on the weighted sum of completion times subject to the precedence constraints. This linear programming problem has an exponential number of constraints. It is not practical to solve it except for small problem instances. We develop a column generation technique for solving the linear programming problem.

C. Column Generation

In a column generation procedure, the linear program is first written in terms of a convex combination of an exponential number of columns. This is called the master problem (MP). The master problem is then solved by successive approximation. The approximation of the master problem is done by restricting the set of columns in the linear programming problem. This is called the restricted master problem (RMP). The dual solution to the restricted master problem is used to verify if the current solution is optimal. If not, a new column is generated to be included in the restricted master problem. This process is repeated until optimality is reached. The practicality of the column generation procedure depends on whether optimality verification and new column generation can be done efficiently. In our case, this column generation procedure is very easy to solve since it reduces to a simple sorting procedure.

We now give a more detailed view of the column generation procedure for the scheduling problem on hand. The master problem (MP) in terms of the extreme points of the polyhedrons $\mathcal{P}_p$ is the following:

$$Z^{MP} = \min \sum_{u \in T} w_u C_u \tag{2}$$

subject to

$$C_u \geq \sum_{k=1}^{n_p!} \lambda_{p(u)}^k \; E_{p(u)}^k(u) \qquad \forall u \tag{3a}$$

$$\sum_{k=1}^{n_p!} \lambda_p^k = 1 \; \forall p \tag{3b}$$

$$C_v \geq C_u + t_v \qquad \forall (u,v) \in L \tag{3c}$$

Instead of using all the extreme points of the polyhedron, the restricted master problem is formulated over a subset of extreme points. Let $Z(p)$ denote a subset of extreme points for processor p. Initially we generate one extreme point for each processor. This is done as follows: For processor p, we randomly order the tasks in J_p and compute the completion time for all the tasks. This vector of completion times is an extreme point of $\mathcal{P}_p$ and is included in the restricted master problem. Therefore, initially $Z(p)$ contains only one element for each processor p. The restricted master problem is similar to the master problem except that it is formulated over the extreme points in $Z(p)$.

$$Z^{RMP} = \min \sum_{u \in T} w_u C_u \tag{4}$$

subject to

$$C_u \geq \sum_{k \in Z(p(u))} \lambda_{p(u)}^k \; E_{p(u)}^k(u) \qquad \forall u \tag{5a}$$

$$\sum_{k \in Z(p)} \lambda_p^k = 1 \qquad \forall p \tag{5b}$$

$$C_v \geq C_u + t_v \qquad \forall (u,v) \in L \tag{5c}$$

Note that $Z^{RMP} \geq Z^{MP}$. We now write the dual to the master problem which we call the column generator (CG):

$$Z^{CG} = \max \; \sum_p \delta_p + \sum_v t_v \sum_{u:(u,v)\in L} \pi_{uv} \tag{6}$$

subject to

$$\theta_u - \sum_{v:(u,v)\in L} \pi_{uv} + \sum_{v:(v,u)\in L} \pi_{vu} \leq w_u \qquad \forall u \tag{7a}$$

$$\delta_p - \sum_{u \in J_p} E_p^k(u)\theta_u \leq 0 \;\; \forall k \qquad \forall p \tag{7b}$$

By linear programming duality we know that $Z^{CG} = Z^{MP}$. Moreover any *feasible* solution to the dual is a lower bound on Z^{MP}. If we solve the restricted master problem, and obtain the optimal dual variables then we can check if that solution is feasible to CG. If it is, then the solution is optimal to the master problem. The dual optimal solution to the restricted master problem will be feasible to equations Equation (7a). However, it may not be feasible to equations Equation (7b) since the RMP uses only a subset of the extreme points. Given a current dual optimal solution $(\delta_p^*, \theta_u^*, \pi_{uv}^*)$ to the RMP, we have to check if

$$\delta_p^* \leq \sum_{u \in J_p} E_p^k(u)\theta_u^* \;\; \forall k \; \forall p.$$

This is equivalent to the following linear programming problem.

$$Z_p(\theta^*) = \min_{C_u \in \mathcal{P}_p} \sum_{u \in J_p} \theta_u^* C_u \tag{8}$$

This is just the classical problem of minimizing the weighted completion time of the tasks on processor p. The weight of task u is θ_u^* in this case. The solution procedure is called Smith's Rule [9] and is the following: Sort the tasks such that

$$\frac{\theta_1^*}{t_1} \geq \frac{\theta_2^*}{t_2} \geq \ldots \geq \frac{\theta_{n_p}^*}{t_{n_p}}$$

and scheduling the tasks in the order of increasing index. In this case, the completion time of task u is $C_u^* = \sum_{v=1}^{u} t_v$ and

$$Z_p(\theta^*) = \sum_{u=1}^{n_p} \theta_u^* C_u^*. \tag{9}$$

Note that C_u^* defines an extreme point of the polyhedron $\mathcal{P}_p$. If $\delta_p^* \leq Z_p(\theta)$, for all p then the current value of $Z^{RMP} = Z^{MP}$. If there is a p such that $\delta_p^* > Z_p(\theta)$ then extreme point corresponding to this processor is added to the restricted master problem and the restricted master is solved again. Given the current dual variable values of θ^* and π^*, we can derive a feasible dual solution as follows: Equation (7a) is satisfied by any dual solution to the RMP. We know that setting $\delta_p = Z_p(\theta^*)$ makes a feasible dual solution. Therefore,

$$\sum_p Z_p(\theta^*) + \sum_v t_v \sum_{u:(u,v)\in L} \pi_{uv}^* \tag{10}$$

is a lower bound on Z^{MP}. The lower bound may not be monotonically increasing. Therefore, we keep track of the current best (highest) lower bound as LB. Instead of solving the MP to optimality, given some threshold ϵ we can terminate computation if the ratio of the solution to the restricted master problem to the best lower bound is less than $(1+\epsilon)$. Algorithm 1 summarizes the column generation procedure.

Algorithm 1 Column Generation

```
1: For each processor $p$ generate $E_p^1$.
2: repeat
3:     Solve RMP and obtain the dual variables.
4:     for all processors $p$ do
5:         Solve CG for $p$ and compute $Z_p(\theta^*)$ as in Equation (9)
6:         if $(Z_p(\theta^*) < \delta_p^*)$ then
7:             Add extreme point to RMP
8:             Update lower bound $LB$ as in Equation (10)
9: until No new extreme points or $Z^{RMP} < (1+\epsilon)LB$
```

III. MARES: A MapReduce Scheduler

Solving the linear program MP defined in the last section gives a lower bound on the min weighted sum schedule. We outline how to convert this (infeasible) linear programming solution to a feasible solution to the scheduling problem that is guaranteed to be within a constant factor of the lower bound (and hence within a constant factor of the optimal solution). In general, it is not possible to convert the linear programming solution into a constant factor approximation algorithm for the scheduling problem. However, the specific structure of the precedence graph for the MapReduce problem leads to a constant factor approximation algorithm. In this section, we focus on the MapReduce problem where the shuffle phase is not the bottleneck. This reduces the problem to a two-phase structure. The constant factor approximation algorithm will be called MapReduce Scheduler (MARES). Leaving out the shuffle phase is done for two reasons. First, the ideas in MARES are used to derive the more complex algorithm in the next section when the shuffle operation in introduced into the picture. Second, there are instances when shuffle is not a significant bottleneck and in this case the algorithm developed here could be used to schedule the tasks. In order to keep the formulation of the problem consistent with the model in the last section, we introduce dummy tasks into the model. Each dummy task is assigned to its own dummy processor. This dummy processor is not part of the linear programming formulation. Dummy tasks will introduce new precedence constraints into the problem and these constraints are taken into account in the linear programming formulation. We introduce two dummy tasks for each job j :

- Start time dummy task S_j that takes r_j unit of time. There is a link in the precedence graph from S_j to all the map tasks for job j.
- Finish time dummy task F_j whose processing time is zero and there is a link from every reduce task of job j to F_j in the precedence graph.

With these two modifications, an example of the precedence graph of a job j is shown in Figure 1. In this example job j has 3 map tasks—1, 2 and 3, and 2 reduce tasks—4 and 5. The objective of the linear programming problem is to minimize the weighted sum of the completion time of the finish time tasks, i.e., $\min \sum_j w_j C_{F_j}$. (The value of $w_u = 0$ for all $u \notin F_j$). The linear programming relaxation of the scheduling problem can be solved to get a lower bound on the optimal solution value. We use the linear programming solution to get a feasible solution that is within a factor of 8 of the lower

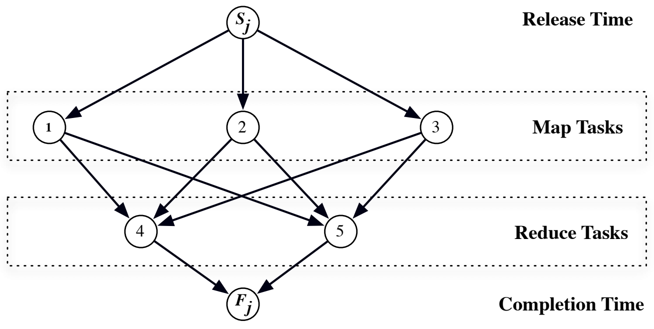


Fig. 1: Task Precedence of a Job

bound, hence giving an 8-approximation algorithm. The basic idea in MARES is the following:

- Task u becomes available only at its LP completion time C_u^{LP}.
- Tasks are scheduled in increasing order of LP completion time as long as all the predecessors of the task have been completed.
- If the predecessor of a task are not completed, it waits until its predecessors complete and then is scheduled in the order of its LP completion time.

Note that the LP completion time is used for determining the available time of the job as well as the scheduling order. The algorithm MARES is outlined in Algorithm 2. In the description, it is assumed that time is slotted and the decisions are made in each time slot. It is easy to make this process polynomial time by advancing time by the task processing times.

Algorithm 2 MARES

```
1: Solve the LP relaxation and obtain $C_u^{LP}$ for all tasks.
2: $A_u \leftarrow C_u^{LP}$ is the available time of $u$.
3: for each time slot $t$ do
4:     for all processors $p$ do
5:         if $p$ is not busy then
6:             Schedule available $u \in J_p, A_u \leq t$ with the lowest $C_u^{LP}$
```

We now outline the proof of the performance guarantee of the MARES. The proof exploits the special structure of the precedence constraints for mapreduce in order to derive the approximation ratio. The approximation results will not hold if the precedence relations are arbitrary. We use M_j to denote the set of map tasks for job j and R_j to denote the set of reduce tasks for job j.

Theorem 1: Let Z^{MP} and Z^H denote the objective function value of the MP linear program and MARES respectively. Then

$$Z^H \leq 8\ Z^{MP}.$$

Proof: Let C_u^H denote the finish completion time of task u on processor $p(u)$. Since tasks are scheduled in the order of LP completion times, once a map task u becomes available on a particular processor, the only tasks that can be processed on the processor before u are those whose LP completion time is not greater than C_u^{LP}. We define

$$D(u) = \{v:\ v \in J_{p(u)},\ C_v^{LP} \leq C_u^{LP}\}.$$

Therefore, for $u \in M_j$, we have

$$C_u^H \le C_u^{LP} + \sum_{v \in D(u)} t_v \tag{11}$$

And from Equation (1) we have

$$\sum_{v \in D(u)} t_v C_v^{LP} \ge f(D(u), p(u)) \ge \frac{1}{2}\left(\sum_{v \in D(u)} t_v\right)^2 \tag{12}$$

Using the fact that $C_u^{LP} \ge C_v^{LP}$ for all $v \in D(u)$, we can write

$$C_u^{LP} \sum_{v \in D(u)} t_v \ge \frac{1}{2}\left(\sum_{v \in D(u)} t_v\right)^2 \tag{13}$$

Therefore $\sum_{v \in D(u)} t_v \le 2\ C_u^{LP},\ \forall u \in M_j$ for all jobs j. Applying this to Equation (11) gives

$$C_u^H \le 3\ C_u^{LP},\ \forall u \in M_j. \tag{14}$$

We use $C_{M_j}^H = \max_{u \in M_j} C_u^H$ to denote the completion time of all the map tasks belonging to job j. There is a key difference between scheduling map and reduce tasks. A map task u can be scheduled as soon at it becomes available at its LP completion time C_u^{LP}. A reduce task u, on the other hand, even if it becomes available at C_u^{LP}, can only be scheduled after all its predecessor map tasks are completed. In other words, a reduce task $u \in R_j$ can be scheduled only after $C_{M_j}^H$. Once all the predecessors of a reduce task are completed, all jobs that will be processed before the reduce task will have LP completion time lower than the reduce job. This statement is almost true except for the case where a task w with $C_w^{LP} > C_u^{LP}$ is already in process on the processor $p(u)$ when the predecessors of $u \in R_j$ are completed. In this case, even if the reduce job has a lower LP completion time, due to non-preemption, the task w will be completed before the reduce job is started. Therefore, we can write for all $u \in R_j$,

$$C_u^H \le C_{M_j}^H + \sum_{v \in D(u)} t_v + t_w \tag{15}$$

where $C_w^{LP} > C_u^{LP}$ and $t_w \le C_w^{LP} \le C_{M_j}^H$. The reason we have $C_w^{LP} \le C_{M_j}^H$ is because task w should have become available before the map completion time since it was in process when the map job completed. Use similar analysis as for the map jobs, we have $\sum_{v \in D(u)} t_v \le 2\ C_u^{LP}$ and $C_{M_j}^H \le 3\ C_{M_j}^{LP}$. With $t_w \le C_w^{LP} \le C_{M_j}^H$, Equation (15) becomes

$$\begin{aligned} C_u^H \le\ & C_{M_j}^H + 2\ C_u^{LP} + C_{M_j}^H \\ \le\ & 6\ C_{M_j}^{LP} + 2\ C_u^{LP} \\ \le\ & 8\ C_u^{LP} \end{aligned}$$

where the last inequality follows from the fact that if $v \in M_j$ and $u \in R_j$, then $C_v^{LP} \le C_u^{LP}$ since this will be a constraint in the linear program. This implies $Z^H \le 8\ Z^{LP}$. ■

A. H-MARES: A Heuristic Implementation of MARES

The idea of making a job available at its LP completion time is mainly to bound the worst case performance. 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. The description of the algorithm is exactly the same MARES except that $A_u = r_j$ if $u \in M_j$ and $A_u = 0$ otherwise. Recall that r_j is the time at which job j enters the system. If all jobs are available at time zero then $A_u = 0$ for all tasks u. The solution of the LP and the scheduling is done exactly as in MARES. We show that H-MARES outperforms MARES in all the experiments performed. However it does not seem straightforward to prove any theoretical performance guarantee for H-MARES. In the performance evaluation section, we show that H-MARES does very well in practice and on the average is within a factor of 1.5 of the lower bound.

IV. MASHERS: A Map-Shuffle-Reduce Scheduler

We now address the problem of jointly scheduling map, shuffle and reduce operations. Before we give a description of the scheduling algorithm we first outline the constraints imposed by the shuffle process. When a map task for a job is completed, the result of this task has to be sent to all the reduce tasks for the job. Some reduce tasks may be on the same processor as the map task while others may be on different processors. Each processor is assumed to have an egress port that is used to send out data and an ingress port to receive data. We assume that once data transfer between two processor begins it cannot be interrupted. We also assume that at most one file can be transmitted or received at any given time instant for any processor. It is possible to relax this assumption to make multiple tasks share the shuffle link but the analysis becomes more complex. Unlike map and reduce tasks, shuffle tasks have to simultaneously possess resources at the sending and receiving side in order to complete the transmission. Therefore shuffle processing can be viewed as an edge scheduling algorithm. Further, the shuffle process is done after the map process and this precedence constraint before the shuffle phase makes the analysis more complex and the competitive ratio is looser than just scheduling when multiple resources are needed. to complete a task. In order to make the formulation compatible with the model in Section II, we model the shuffle process as follows:

- Corresponding to processor p in the system, we add two more processors $I(p)$ and $O(p)$ where $I(p)$ represents the ingress port at processor p and $O(p)$ represents the egress port. Each of these additional processors is treated like regular a processor in the LP formulation.
- In addition to the start and finish nodes for each job as in Section III, we introduce $2|M_j||R_j|$ additional nodes where a pair of nodes represent the two ends of the transfers.
- Each map node has $|R_j|$ transfer nodes associated with it and each reduce node has $|M_j|$ transfer nodes associated with it. Each map task precedes its corresponding transfer tasks and each reduce task succeeds its corresponding transfer task. The map part of the transfer precedes

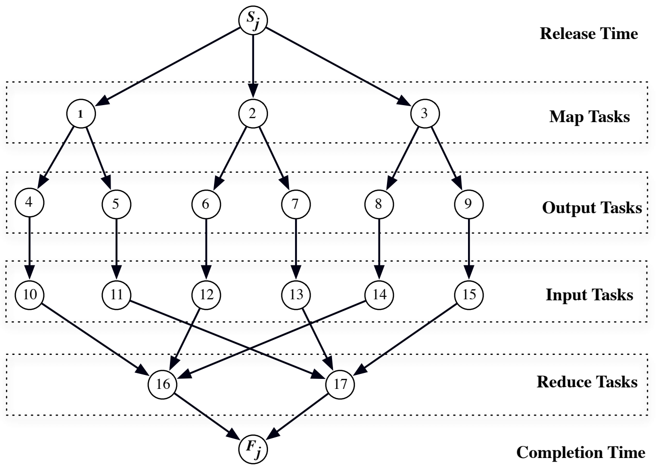


Fig. 2: Task Precedence of a Job with Data Shuffle

the reduce part of the transfer. The processing time of the map side transfer equals the actual transfer time between the corresponding map and reduce processors; the processing time of the reduce side transfer is set to be zero.

- The fact that two resources have to be held simultaneously is not considered in the linear program and is only introduced in the scheduling algorithm.
- The completion time of the two tasks that are at the two ends of a transfer are set equal.

An example of this new precedence graph is shown in Figure 2. In the example, task 1, 2 and 3 are map tasks, task 16 and 17 are reduce tasks. Tasks from 4 to 9 are outgoing tasks and task from 10 to 15 are the incoming tasks. In Figure 2, the completion time of task 9 and task 15 are set equal since they represent a transfer. The same statement can be made for all the transfer task pairs. With this new precedence graph, the LP formulation stays the same as in Section II. The LP is solved using the column generation procedure and all the LP completion times are obtained.

A. An outline of MASHERS

We first give a high level view of MASHERS Algorithm 3:

- Once the lower bound LP is solved, the map and reduce tasks are scheduled as Algorithm 2. A reduce task has to wait until all its predecessors are completed.
- The Shuffle tasks are viewed as edge scheduling problem which schedules the shuffle edges in the precedence graph. This is done by partitioning the edges into groups using Algorithm 4 and scheduling the groups in order. Let G_0 denote the original precedence constrained graph; let G_i and G'_i denote the edge groups partitioned in the i the iteration of Algorithm 4. Within a group, edges are scheduled in the order of their LP completion time.
- Let SH denote the set of shuffle edges in the precedence graph. In Figure 2, edge $(9,15)$ is a shuffle edge that transfers the output of map task 3 to the reduce task 17. We extend the definition of the completion time as follows: We define the completion time of a *shuffle edge* $e=(u,v)$ in the linear programming solution as the completion time of its nodes, that is, $C_e^{LP}=C_u^{LP}=C_v^{LP}$.
- We assume time is slotted and the scheduling algorithm is executed in each time slot. In the description of the algorithm MASHERS we assume that algorithm PARTITION has already partitioned the shuffle edges.

Algorithm 3 MASHERS

```
1: Solve the LP and obtain $C_u^{LP}$ for all tasks.
2: Execute Algorithm 4 and obtain $G'_1, G'_2, ...G'_k$
3: $A_u \leftarrow C_u^{LP}$, $A_{e(u,v)} \leftarrow C_u^{LP}$ for $e=(u,v)\in SH$.
4: for each time slot $t$ do
5:     Find the largest $i$ such that $G'_i$ has unscheduled edges.
6:     $ES_i \leftarrow \{e : e \in E'_i,\ A_e \le t$, $e$ is free and available$\}$
7:     while $ES_i \neq \emptyset$ do
8:         Schedule $e(u,v) \in ES_i$ with the smallest $C_e^{LP}$
9:     for all processors $p$ do
10:        if $p$ is not busy then
11:            Schedule $u \in J_p, A_u \le t$ with the lowest $C_u^{LP}$ whose
               predecessors are completed
```

We now describe the partitioning process.

B. Partitioning the Shuffle Edges

The partition algorithm groups the shuffle edges. The key idea is to construct the groups such that edges with similar processing times and LP completion times belong to the same group. In the description of Algorithm 4 we use the following additional notation. Given a graph H, $m(H)=\max_{e\in H} C_e^{LP}$ denote the maximum LP completion time of an edge in H. Given some set of edges S, let $p(S)$ denote sum of the processing times of the edges in S. Let $\phi(G_i)=3m(G_i)+2p(G_i)$.

Algorithm 4 PARTITION

```
1: Sort edges $E$ in $G_0$ in non-decreasing order of $C_e^{LP}$.
2: $i \leftarrow 1$
3: repeat
4:     $E_i \leftarrow \emptyset$
5:     for all $e \in E_{i-1}$ in the sorted order do
6:         if $\phi(G_i) \le \frac{1}{2}\phi(G_{i-1})$ holds by adding $e$ to $E_i$ then
7:             add $e$ to $E_i$
8:         else
9:             add $e$ to $E'_i$
10:    $i \leftarrow i+1$
11: until $E_i$ is empty
```

The partitioning and scheduling algorithm works as follows:

- The algorithm first partitions G_0 into G_1 and G'_1 with respect to $\phi(G_1) \le \frac{1}{2}\phi(G_0)$. Then it iteratively partitions G_{i-1} into G_i and G'_i until G_i is empty.
- All edges in G'_i are scheduled before considering edges in G'_{i-1}.

C. MASHERS: Performance Guarantee

In this section, we show that MASHERS gives a constant factor competitive ratio. This is done by first analyzing the performance of the shuffle operation and then incorporating the map and reduce into the analysis. The partition process is similar to the partition operation in [8]. The potential function $\phi(G_i)$ used for partitioning in [8] just involves the processing time (the LP completion times are only used for ordering). For our problem, the potential function used for partitioning is a weighted sum of the LP completion time of the edges and the total processing time at the node. This is done in order to include the performance of the map operation in the overall analysis. We now give preliminary result that is used to bound the performance guarantee of MASHERS.

Lemma 2: For $e \in E'_i$,

$$C_e^H \le 2\phi(G_{i-1}) \quad (16)$$

Proof: We prove this by induction. Let $N_{G_i}(u)$ denote the set of edges incident to $u \in G_i$. $p(N_{G_i}(u))$ is the sum of transfer times in $N_{G_i}(u)$. For an edge $e = (u, v)$ in graph G'_k, we have

$$\begin{aligned} C_e^H &\le C_u^H + p(N_{G'_k}(u)) + p(N_{G'_k}(v)) \\ &\le C_u^H + 2p(G'_k) = C_u^H + 2p(G'_{k-1}) \\ &\le 3C_u^{LP} + 2p(G'_{k-1}) \le \phi(G_{k-1}). \end{aligned}$$

Now we will prove that if we can schedule all edges in $E'_k, E'_{k-1}, ..., E'_{i+1}$ within $2\phi(G_i)$, then all edges in $E'_k, E'_{k-1}, ..., E'_i$ can be scheduled within $2\phi(G_{i-1})$.

For $e = (u, v) \in E'_i$, it needs to wait for at most $2\phi(G_i)$ time before the set E'_i is considered. In addition, it can take $C_u^H + p(N_{G'_i}(u)) + p(N_{G'_i}(v))$ to complete. That is,

$$C_e^H \le 2\phi(G_i) + C_u^H + p(N_{G'_i}(u)) + p(N_{G'_i}(v)) \quad (17)$$

Since we have $\phi(G_i) \le \phi(G_{i-1})/2$ and $C_u^H + p(N_{G'_i}(u)) + p(N_{G'_i}(v)) \le 3C_e^{LP} + 2p(G_{i-1}) \le \phi(G_{i-1})$,

$$C_e^H \le 2\phi(G_{i-1})/2 + \phi(G_{i-1}) = 2\phi(G_{i-1}) \quad (18)$$

This proves the lemma. ■

Theorem 3: Let Z^{LP} and Z^H denote the objective function value of LP and the sum of completion time by Algorithm 4 respectively. Then

$$Z^H \le 58\ Z^{LP}.$$

Proof: As in the case of MASHERS, for $u \in M_j$,

$$C_u^H \le 3\ C_u^{LP} \quad (19)$$

An edge $e = (u, v) \in E'_i$ is added to G'_i, because it could not be added to G_i. Let $p(u)$ denote the sum of transfer times of edges incident to u in G_i whose LP completion times are less than C_e^{LP}. Since edges are added in the order of LP completion times and e could not be added to G_i implies that

$$3C_e^{LP} + 2p(u) > \frac{1}{2}\phi(G_{i-1}).$$

(In the above expression we assume, without loss of generality that node u was the blocking node.) Let $D(u)$ denote the set of edges incident to u in G_0 and with smaller LP completion time than e, then $p(D(u)) \ge p(u)$. Together with $C_e^{LP} \ge p(D(u))/2$ from the LP we have,

$$\frac{1}{2}\phi(G_{i-1}) \le 3C_e^{LP} + 2p(D(u)) \le 7C_e^{LP}$$

Combine it with Equation (16), we have

$$C_e^H \le 28\ C_e^{LP} \quad (20)$$

For reduce tasks, similar to the proof in Section III, for any $v \in R_j$

$$\begin{aligned} C_v^H &\le C_v^H + \sum_{u \in D(v)} t_u + t_w && (21) \\ &\le 28\ C_v^{LP} + 2\ C_v^{LP} + 28\ C_v^{LP} && (22) \\ &\le 58\ C_v^{LP} && (23) \end{aligned}$$

This implies $Z^H \le 58\ Z^{LP}$. ■

There are two comments in order here:

- It is possible to tighten the analysis to get a better approximation ratio but we do not do that here in order keep the analysis simple.
- We use MASHERS to develop two heuristic algorithms D-MASHERS and H-MASHERS that we evaluate in the next section.

D. Heuristics D-MASHERS and H-MASHERS

The main reason for partitioning the shuffle edges is to guard against the worst case where a edge with a long processing time is scheduled early and it delays a large number of tasks. Though MASHERS provides a worst case competitive guarantee, its performance suffers in practice since it guards against corner cases. If the transfer times are not too different then we can skip the partitioning process and just use the LP solution to guide which task to schedule. We can use two different variants:

- D-MASHERS: A task is delayed by its LP completion time and is scheduled in the order of LP completion times as long as all its predecessors are done.
- H-MASHERS: Tasks are not delayed by their LP completion times and are scheduled in the order of LP completion times as long as its predecessors are done.

In the experimental section, we evaluate both D-MASHERS and H-MASHERS.

V. Performance Evaluation

In this section, we present a simulation based evaluation of the algorithms developed in this paper.

A. Simulation Setting

The workloads of real MapReduce system are not publicly available. Therefore, we use a synthetic workload generation model similar to [2]. Given the number of jobs and processors, we generate a set of map and reduce tasks for each job. The processing time of a task is uniformly distributed in $[1, 10]$ unit. Between a map task of size m and a reduce task of size r, the data shuffle delay is set to be $(m + r)/3$. For n jobs, their weights are uniformly distributed in $[1, n]$ and their release times are uniformly distributed in $[1, 20]$. Once we generate a problem instance, we pass its workload information to our scheduler.

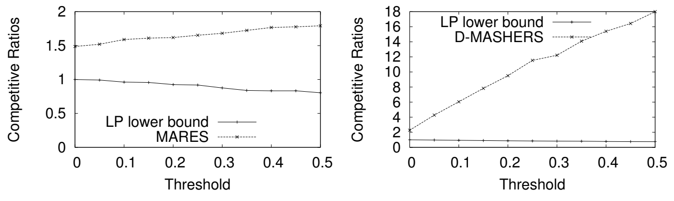
We assume that the processing time and shuffle delays are known to the scheduler. In the online case where jobs arrive over time and their information is known only when they arrive, we update job information by adding new jobs and remove finished tasks at the moment a job arrives. Then we run our scheduler again. Furthermore, we may also adjust the existing sets of extreme points for the new set of jobs, so that our scheduler does not have to run from the beginning. However, this is not the main focus of this paper and is left for future work.

B. LP Stopping Threshold Test

As mentioned in Section II, for large problem instances, we may stop the Column Generation once the ratio of the master problem to the best LP lower bound becomes less than a threshold $1 + \epsilon$. For the purpose of simulating some large problem instances, we need to test the effect of different ϵ's on the LP lower bound and our heuristics and pick a suitable ϵ value.

In each test, first we solve the LP optimal as reference point. Then for the same problem instance, we vary ϵ and solve the problem with our heuristics using the degraded LP solution. For the two phase problem, we use 50 processors and 30 jobs. Each job has 20 map tasks and 10 reduce tasks on average. For the three phase problem, we use 30 processors and 20 jobs. Each job has 8 map tasks and 3 reduce tasks on average. Then for each ϵ value, we run 10 trials. The results in Figure 3 are competitive ratios (results over the LP optimal) averaged in these 10 trails.

The threshold does not affect the performance of either the LP or the MARES as shown in Figure 3a. We only see a slight decrease/increase in the LP lower bound/MARES competitive ratio, even when $\epsilon = 0.5$. This is not true for the three phase map-shuffle-reduce problem. In Figure 3b, although the LP lower bound does not decrease much with a large threshold, the competitive ratio of D-MASHER increases dramatically. Therefore, in the following tests, we always set $\epsilon = 0.5$ for the two phase but $\epsilon = 0$ for the three phase.



(a) MARES (b) MASHERS

Fig. 3: Threshold ϵ Test

In the rest of the tests, for each test, we vary the number of jobs to generate different sizes of the problem. For a particular number of jobs, we run 20 trials and plot them against either the optimal or the LP lower bound.

C. Evaluation of Two Phase Map-Reduce Heuristics

For the two phase problem, we implement an Integer Program (IP). With small problem instances, we may obtain the optimal solution in reasonable time. Figure 4a shows the competitive ratios for only 30 processors and from 5 to 16 jobs; each job has 10 map tasks and 3 reduce tasks on average. As we can see, H-MARES is the closest to the optimal and the LP lower bound is closer than MARES, whose competitive ratio is about 1.5 to 2.

With large problem instances with 100 processors and from 5 to 50 jobs (30 map tasks and 10 reduce tasks per job), we do not calculate the optimal solution. However, we can still see in Figure 4b the competitive ratios of MARES and H-MARES are far away from MARES' approximation guarantee of 8.

D. Evaluation of Map-Shuffle-Reduce Heuristics

We did similar tests for the three phase problem except that we do not have an IP to obtain the optimal solution but we compre the heuristic results to the LP lower bound. With 30 processors and from 5 to 40 jobs (each has 8 map tasks and 3 reduce tasks on average), however, we can still see in Figure 5a, the two heuristics generally achieve competitive ratio less than 3.

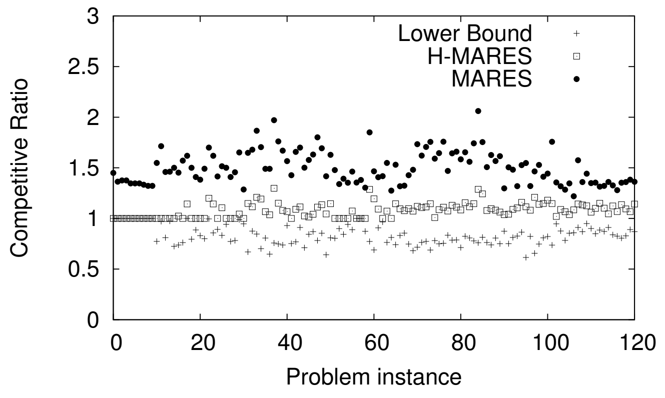
In Figure 5b we plot the average competitive ratio as we increase the number of jobs. It is interesting to see that when there are fewer jobs, H-MASHER beats D-MASHER, for it does not hold a task. When there are more and more job the performance of D-MASHERS improves when compared to H-MASHERS. This seems to indicate that hold a task until its LP completion time actually improves the performance of the heuristic. One possible explanation for this phenomenon is that in larger problems there is a wider range of processing time values and not holding long tasks back hurts the completion time of all tasks that come after it.

VI. Related Work

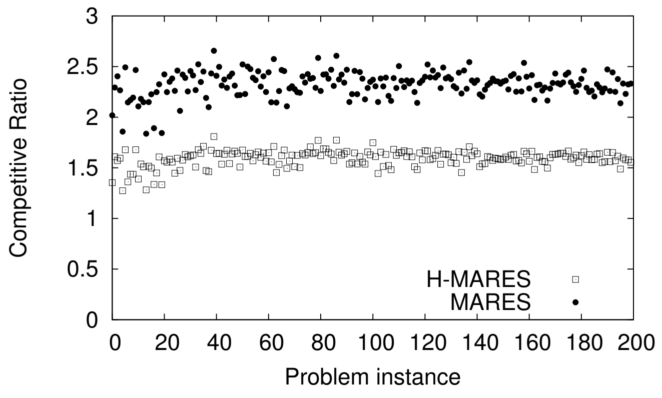
Job scheduling on parallel machines is a well-studied problem, refer to [9] for more details. In particular, our work is related to the problem of machine scheduling with precedence constraints. The scheduling problem with a general precedence graph is among the most difficult problems in the area of machine scheduling [10], [11], [12], [13]. For the objective of minimizing the total weighted completion time, the best approximation algorithm is due to Queyranne and Schulz [13]. In the problem that we consider, the precedence between map and reduce induces a depth two precedence graph. However unlike the standard machine scheduling problems, scheduling the shuffle phase resembles the data migration problem. In data migration problems, the migration process is modeled by a transfer graph, in which nodes represent the data transferring entities and edges represent the transfer links. Two edges incident on the same node cannot transfer simultaneously. A node completes when all edges incident on it complete. Kim [8] gave an LP-based 10-approximation for general processing times , which was then improved by Gandhi and Mestre [14] with a combinatorial algorithm. Hajiaghayi *et al.* [15] generalize the technique in [14] and provide a local transfer protocol where multiple transfers can take place concurrently. Other related work is on sum multi-coloring of graphs including [16], [17]. Our work differs from all these papers since the map phase that precedes the shuffle phase makes the analysis different from a traditional data transfer problems. The closest work to our problem is by Chang *et al.* [2], but that paper does not consider the explicit precedence constraints between map and reduce as well a the shuffle phase of the MapReduce problem.

VII. Conclusion

We modeled to end-to-end performance of the MapReduce process and developed constant factor approximation algorithms for minimizing the weighted response time. Based

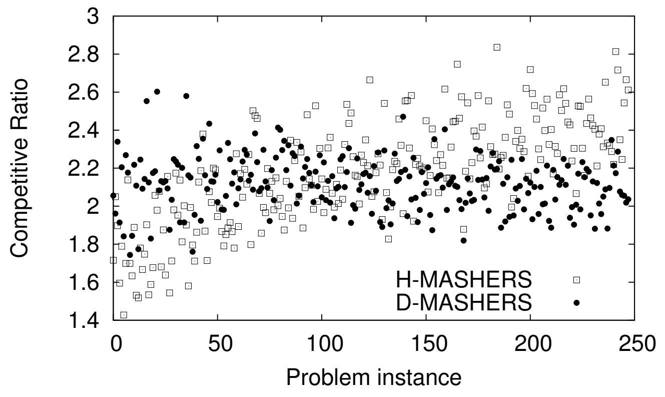


(a) Results againt the Optimal

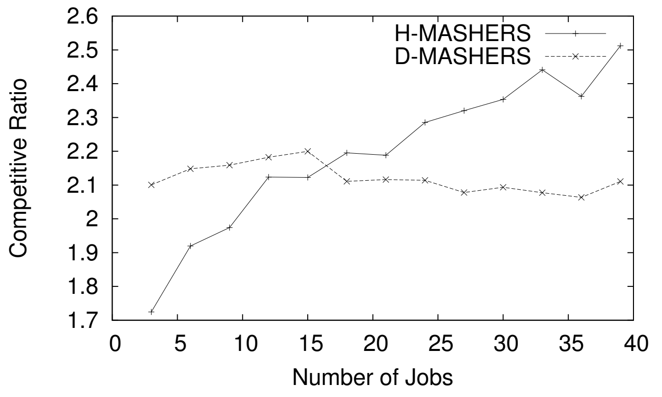


(b) Large problem instances

Fig. 4: Performance test for Two phases Map-Reduce



(a) Competitive Ratios



(b) Average

Fig. 5: Performance test for Map-Shuffle-Reduce

on these guaranteed performance algorithms we developed heuristics that were tested experimentally and performed significantly better that the worst case guaranteed performance. We are currently working on improving the worst case performance guarantees as well as testing the algorithm on larger data sets.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *USENIX OSDI*, 2004.

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

[3] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM*, 2011.

[4] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *SOSP*, 2009.

[5] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, 2010, pp. 265–278.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *USENIX OSDI*, 2010.

[7] J. Lenstra, A. Rinnooy Kan, and P. Brucker, "Complexity of machine scheduling problems," *Annals of Discrete Mathematics*, vol. 1, pp. 343–362, 1977.

[8] Y. Kim, "Data migration to minimize the total completion time," *Journal of Algorithms*, vol. 55, no. 1, pp. 42–57, 2005.

[9] P. Brucker, *Scheduling algorithms*. Springer, 2004.

[10] R. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[11] J. Lenstra and A. Kan, "Complexity of scheduling under precedence constraints," *Operations Research*, pp. 22–35, 1978.

[12] S. Chakrabarti, C. Phillips, A. Schulz, D. Shmoys, C. Stein, and J. Wein, "Improved scheduling algorithms for minsum criteria," *Automata, Languages and Programming*, pp. 646–657, 1996.

[13] M. Queyranne and A. 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.

[14] R. Gandhi and J. Mestre, "Combinatorial algorithms for data migration to minimize average completion time," *Algorithmica*, vol. 54, no. 1, pp. 54–71, 2009.

[15] M. Hajiaghayi, R. Khandekar, G. Kortsarz, and V. Liaghat, "On a local protocol for concurrent file transfers," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 269–278.

[16] A. Bar-Noy, M. Halldórsson, G. Kortsarz, R. Salman, and H. Shachnai, "Sum multicoloring of graphs," *Journal of Algorithms*, vol. 37, no. 2, pp. 422–450, 2000.

[17] R. Gandhi, M. Halldórsson, G. Kortsarz, and H. Shachnai, "Improved bounds for scheduling conflicting jobs with minsum criteria," *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 1, p. 11, 2008.