# Scheduling MapReduce Jobs and Data Shuffle on Unrelated Processors

Dimitris Fotakis[1], Ioannis Milis[2], Orestis Papadigenopoulos[1],
Emmanouil Zampetakis[3], and Georgios Zois[2]

[1] School of Electrical and Computer Eng., National Technical University of Athens
fotakis@cs.ntua.gr, opapadig@corelab.ntua.gr
[2] Department of Informatics, Athens University of Economics and Business, Greece
{milis, georzois}@aueb.gr
[3] CSAIL, Massachusetts Institute of Technology, USA, mzampet@mit.edu

**Abstract.** We propose constant approximation algorithms for generalizations of the Flexible Flow Shop (FFS) problem which form a realistic model for non-preemptive scheduling in MapReduce systems. Our results concern the minimization of the total weighted completion time of a set of MapReduce jobs on unrelated processors and improve substantially on the model proposed by Moseley et al. (SPAA 2011) in two directions: (i) we consider jobs consisting of multiple Map and Reduce tasks, which is the key idea behind MapReduce computations, and (ii) we introduce into our model the crucial cost of the data shuffle phase, i.e., the cost for the transmission of intermediate data from Map to Reduce tasks. Moreover, we experimentally evaluate our algorithms compared with a lower bound on the optimal cost of our problem as well as with a fast algorithm, which combines a simple online assignment of tasks to processors with a standard scheduling policy, and performs better for instances where the processing times of Map and Reduce tasks are drawn from the same uniform distributions.

## 1 Introduction

The widespread use of MapReduce [7] to implement massive parallelism for data intensive computing motivates the study of new challenging shop scheduling problems. Indeed, a MapReduce job consists of a set of Map tasks and a set of Reduce tasks that can be executed simultaneously, provided that no Reduce task of a job can start execution before all the Map tasks of this job are completed. Moreover, a significant part of the processing cost in MapReduce applications is the communication cost due to the transmission of intermediate data from Map tasks to Reduce tasks (a.k.a. data shuffle, see e.g., [2, 1]). To exploit the inherent parallelism, the scheduler, which operates in centralized manner, has to efficiently assign and schedule Map and Reduce tasks to the available processors. In this context, standard shop scheduling problems are revisited to capture key constraints and singularities of MapReduce systems. In fact, a few results have been recently proposed based on simplified abstractions and resulting in known variants of the classical Open Shop and Flow Shop scheduling problems [4, 5, 13].

In this work, we significantly generalize the Flexible Flow Shop (FFS) model for MapReduce scheduling proposed in [13]. Recall that in the FFS problem, we are given

a set of jobs, each consisting of a number of tasks (each task corresponds to a stage), to be scheduled on a set of parallel processors dedicated to each stage. The jobs should be executed in the same fixed order of stages, without overlaps between different stages of the same job. Our generalization extends substantially the model proposed in [13] by taking into account all the important constraints of MapReduce systems: (a) each job has multiple tasks in each stage; (b) the assignment of tasks to processors is flexible; (c) there are dependencies between Map and Reduce tasks; (d) the processors are unrelated to capture data locality; and (e) there is a significant communication cost for the data shuffle. Our goal is to find a non-preemptive schedule minimizing the standard objective of total weighted completion time for a set of MapReduce jobs.

**Related Work.** Known results for the FFS problem concern the two-stage case on parallel identical processors. For the makepsan objective a PTAS is known [14], while for the objective of total weighted completion time, a simple 2-approximation algorithm was proposed in [9] for the special case where each stage has to be executed on a single processor. For the latter case, [13] recently proposed a QPTAS which becomes a PTAS for a fixed number of task processing times.

In the MapReduce context, most of the previous work concerns the experimental evaluation of scheduling heuristics, from the viewpoint of finding good tradeoffs between different practical criteria (see e.g., [16]). From a theoretical viewpoint, all known results [4, 5, 13] concern the minimization of total weighted completion time. Chang et al. [4] studied a simple model, equivalent to the well-known *concurrent open shop* problem [12], where there are no dependencies between Map and Reduce tasks and the assignment of tasks to processors is given. Chen et al. [5] generalized the last model by considering dependencies between Map and Reduce tasks and presented an LP-based $8$-approximation algorithm. Moreover, they managed to incorporate the data shuffle into their model and to derive a $58$-approximation algorithm. Finally, Moseley et al. [13] suggested the connection of MapReduce scheduling to the FFS problem and proposed a $12$-approximation algorithm, for the case of identical processors, and a $6$-approximation algorithm for the very restricted case of unrelated processors where each job has a single Map and a single Reduce task. For both cases they also proposed constant competitive online algorithms with constant speed augmentation.

**Our Results and Contributions.** We present constant approximation algorithms which substantially generalize the results of [13] for MapReduce scheduling on unrelated processors in two directions motivated by practical applications of MapReduce systems. In fact, we deal with jobs consisting of multiple Map and Reduce tasks and also incorporate the shuffle phase into our setting. As it has been observed in [13], new ideas and techniques are required for both these directions.

In Section 2, we present a $54$-approximation algorithm for the Map-Reduce scheduling problem when jobs consist of multiple Map and Reduce tasks. We first give an interval-indexed LP-relaxation for the problem of minimizing the total weighted completion times separately for Map and Reduce tasks on unrelated processors. Our LP-relaxation is inspired by that proposed by Hall et al. [10] for scheduling a set of single task jobs on unrelated processors under the same objective. However, in our setting, only the task finishing last (instead of all tasks) contributes to the objective value, which complicates the analysis. Recently, Correa et al. [6] proposed a similar LP-relaxation

for a more general problem, where, instead of jobs consisting of tasks, they have a set of job orders and the completion time of each order is specified by the completion of the job finishing last. Since scheduling multitask Map and Reduce jobs separately is quite similar to the setting considered in [6], we can apply their approximation result to scheduling the Map and Reduce tasks separately. Next, extending the ideas in [13] for single task jobs, we concatenate the two schedules into a single schedule that respects the task dependencies.

In Section 3, we incorporate the data shuffle phase into our model by introducing an additional set of *Shuffle tasks*, each one associated with a communication cost (expressed as processing time). When the Shuffle tasks are scheduled on the same processors as the corresponding Reduce tasks, we are able to keep the same approximation ratio of $54$ for the Map-Shuffle-Reduce scheduling problem. Moreover, we prove an approximation ratio of $81$ when the Shuffle tasks can be executed on different processors than their corresponding Reduce tasks. To the best of our knowledge, this is the most general setting of the FFS problem (with a special third stage) for which a constant approximation guarantee is known.

In Section 4, we compare experimentally the performance of our LP-based approximation algorithm with a lower bound on the optimal cost of our problem as well as with a simple and fast algorithm. The latter algorithm combines a simple assignment of the tasks, using an online algorithm for makespan minimization on unrelated processors with logarithmic competitive ratio [3], with the standard Weighted Shortest Processing Time first (WSPT) scheduling policy. As we observe, for instances where the processing times of Map and Reduce tasks are drawn from the same uniform distributions, the simple algorithm performs well enough, while, for instances that capture data locality issues, the more sophisticated LP-based algorithm achieves a better performance also in practise. Moreover, we show that the (empirical) approximation ratio of our algorithms is considerably smaller than the corresponding theoretical upper bound. As far as we know, these are the first experimental results for evaluating the performance guarantee of MapReduce scheduling on unrelated processors.

**Problem Statement and Notation.** In the sequel we consider a set $\mathcal{J} = \{1, 2, \ldots, n\}$ of $n$ MapReduce jobs to be executed on a set $\mathcal{P} = \{1, 2, \ldots, m\}$ of $m$ unrelated processors. Each job is available at time zero, is associated with a positive weight $w_j$ and consists of a set of Map tasks and a set of Reduce tasks. Let $\mathcal{M}$ and $\mathcal{R}$ be the set of all Map and all Reduce tasks respectively. Each task is denoted by $\mathcal{T}_{k,j} \in \mathcal{M} \cup \mathcal{R}$, where $k \in N$ is the task index of job $j \in \mathcal{J}$ and is associated with a vector of non-negative processing times $\{p_{i,k,j}\}$, one for each processor $i \in \mathcal{P}_b$, where $b \in \{\mathcal{M}, \mathcal{R}\}$. Let $\mathcal{P}_{\mathcal{M}}$ and $\mathcal{P}_{\mathcal{R}}$ be the set of Map and the set of Reduce processors respectively. For convenience, we assume that $\mathcal{P}_{\mathcal{M}} \cap \mathcal{P}_{\mathcal{R}} = \emptyset$, however we are able to extend our results to the case where the two sets of processors are not necessarily disjoint (or even are identical). Each job has at least one Map and one Reduce task and every Reduce task can start its execution after the completion of all Map tasks of the same job.

For a given schedule we denote by $C_j$ and $C_{k,j}$ the completion times of each job $j \in \mathcal{J}$ and each task $\mathcal{T}_{k,j} \in \mathcal{M} \cup \mathcal{R}$ respectively. Note that, due to the precedence constraints between Map and Reduce tasks, $C_j = \max_{\mathcal{T}_{k,j} \in \mathcal{R}}\{C_{k,j}\}$. By $C_{max} = \max_{j \in \mathcal{J}}\{C_j\}$ we denote the makespan of the schedule, i.e., the completion time of the job which

finishes last. Our goal is to schedule *non-preemptively* all Map tasks on processors of $\mathcal{P}_\mathcal{M}$ and all Reduce tasks on processors of $\mathcal{P}_\mathcal{R}$, with respect to their precedence constraints, so as to minimize the total weighted completion time of the schedule, i.e., $\sum_{j\in\mathcal{J}} w_j C_j$. We refer to this problem as Map-Reduce scheduling problem.

Concerning the complexity of Map-Reduce scheduling problem, it generalizes the FFS problem which is known to be strongly $\mathcal{NP}$-hard [8], even when there is a single Map and a single Reduce task that has to be assigned only to one Map and one Reduce processor respectively.

## 2 The Map-Reduce Scheduling Problem

In this section, we present a $54$-approximation algorithm for the Map-Reduce scheduling problem. Our algorithm is executed in the following two steps: (i) it computes a $27/2$-approximate schedule for assigning and scheduling all Map tasks (resp. Reduce tasks) on processors of the set $\mathcal{P}_\mathcal{M}$ (resp. $\mathcal{P}_\mathcal{R}$) and (ii) it merges the two schedules in one, with respect to the precedence constraints between Map and Reduce tasks of each job. Step (ii) is performed by increasing the approximation ratio by a factor of $4$.

$$\textbf{LP(b)}: \text{minimize} \sum_{j\in\mathcal{J}} w_j C_{D_j}$$

$$\text{subject to}: \sum_{i\in\mathcal{P}_b,\ell\in\mathcal{L}} y_{i,k,j,\ell} \geq 1, \qquad \forall T_{k,j}\in b \tag{1}$$

$$C_{D_j} \geq C_{k,j}, \qquad \forall j\in\mathcal{J}, T_{k,j}\in b \tag{2}$$

$$\sum_{i\in\mathcal{P}_b}\sum_{\ell\in\mathcal{L}} (1+\delta)^{\ell-1} y_{i,k,j,\ell} \leq C_{k,j}, \qquad \forall T_{k,j}\in b \tag{3}$$

$$\sum_{T_{k,j}\in b} p_{i,k,j} \sum_{t\leq\ell} y_{i,k,j,t} \leq (1+\delta)^\ell, \qquad \forall i\in\mathcal{P}_b, \ell\in\mathcal{L} \tag{4}$$

$$p_{i,k,j} > (1+\delta)^\ell \Rightarrow y_{i,k,j,\ell} = 0, \ \forall i\in\mathcal{P}_b, T_{k,j}\in b, \ell\in\mathcal{L} \tag{5}$$

$$y_{i,k,j,\ell} \geq 0, \qquad \forall i\in\mathcal{P}_b, T_{k,j}\in b, \ell\in\mathcal{L}$$

**Scheduling Map Tasks and Reduce Tasks.** To schedule the Map and Reduce tasks separately on the processors $\mathcal{P}_\mathcal{M}$ and $\mathcal{P}_\mathcal{R}$, respectively, we formulate the interval-indexed LP-relaxation above for minizing the total weighted completion time. For notational convenience, we use an argument $b\in\{\mathcal{M},\mathcal{R}\}$ to refer either to Map or to Reduce sets of tasks. We define $(0, t_{\max} = \sum_{T_{k,j}\in b} \max_{i\in\mathcal{P}_b} p_{i,k,j}]$ to be the time horizon of potential completion times, where $t_{\max}$ is an upper bound on the makespan of a feasible schedule. We discretize the time horizon into intervals $[1,1], (1,(1+\delta)], ((1+\delta),(1+\delta)^2], \ldots, ((1+\delta)^{L-1}, (1+\delta)^L]$, where $\delta\in(0,1)$ is a small constant, and $L$ is the smallest integer such that $(1+\delta)^{L-1}\geq t_{\max}$. Let $I_\ell = ((1+\delta)^{\ell-1}, (1+\delta)^\ell]$, for $1\leq\ell\leq L$, and $\mathcal{L}=\{1,2,\ldots,L\}$. Note that, interval $[1,1]$ implies that no job finishes its execution before time 1; in fact, we can assume, w.l.o.g., that all processing times are positive integers. Note also that the number of intervals is polynomial in the size of the instance and in $1/\delta$. For each processor $i\in\mathcal{P}_b$, task $T_{k,j}\in b$ and $\ell\in\mathcal{L}$, we introduce a variable $y_{i,k,j,\ell}$ that indicates if task $T_{k,j}$ is completed on processor $i$ within the time

interval $I_\ell$. Furthermore, for each task $T_{k,j} \in T$, we introduce a variable $C_{k,j}$ corresponding to its completion time. For every job $j \in \mathcal{J}$, we introduce a dummy task $D_j$ with zero processing time processed after the completion of each task $T_{k,j} \in b$. Note that, the corresponding integer program is a $(1+\delta)$-relaxation of the original problem.

Our objective is to minimize the sum of weighted completion times of all jobs. Constraints (1) ensure that each task is completed on a processor of the set $\mathcal{P}_b$ in some time interval. Constraints (2) assure that for each job $j \in \mathcal{J}$, the completion of each task $T_{k,j}$ precedes the completion of task $D_j$. Constraints (3) impose a lower bound on the completion time of each task. For each $\ell \in \mathcal{L}$, constraints (4) and (5) are validity constraints which state that the total processing time of jobs executed up to an interval $I_\ell$ on a processor $i \in \mathcal{P}_b$ is at most $(1+\delta)^\ell$, and that if processing a task $T_{k,j}$ on a processor $i \in \mathcal{P}_b$ takes more than $(1+\delta)^\ell$, $T_{k,j}$ should not be scheduled on $i$, respectively.

Our algorithm, called Algorithm TASKSCHEDULING($b$), starts from an optimal fractional solution $(\bar{y}_{i,k,j,\ell}, \bar{C}_{k,j}, \bar{C}_{D_j})$ to LP(b) and, working similarly to [6, Section 5], rounds it to an integral $27/2$-approximate schedule of the jobs $\mathcal{J}$ on processors $\mathcal{P}_b$. The idea is to partition the set of tasks $T_{k,j}$ into classes $S(\ell) = \{\mathcal{T}_{k,j} \in b \mid (1+\delta)^{\ell-1} \leq a\bar{C}_{k,j} \leq (1+\delta)^\ell\}$, where $\ell \in \{1,\ldots,L\}$ and $a > 1$ is a parameter, according to their (fractional) completion time in the optimal solution of LP(b), and to use [15, Theorem 2.1] for scheduling the tasks in each class $S(\ell)$ independently. In fact, Algorithm TASKSCHEDULING($b$) can be regarded as a generalization of the approximation algorithm in [10, Section 4], where the objective is to minimize weighted completion time, but each job consists of a single task (see also the discussion in [6, Section 5]).

More specifically, we first observe that by the definition of $S(\ell)$ and due to constraints (1) and (3), for each task $T_{k,j} \in S(\ell)$, $\sum_{i\in\mathcal{P}_b}\sum_{t\leq\ell} y_{i,k,j,t} \geq \frac{a-1}{a}$. Otherwise, it would be $\sum_{i\in\mathcal{P}_b}\sum_{t\geq\ell+1} y_{i,k,j,t} > \frac{1}{a}$, which implies $a\bar{C}_{k,j} > (1+\delta)^\ell$. Therefore, if we set $y^*_{i,j,k,t} = 0$, for all $t \geq \ell+1$, and $y^*_{i,j,k,t} = \frac{a}{a-1}\bar{y}_{i,k,j,t}$, for all $t \leq \ell$, we obtain a solution $y^*_{i,k,j,t}$ that satisfies the constraints (1), (4), and (5) of LP(b), if the right-hand side of (4) is multiplied by $a/(a-1)$. Therefore, for each $\ell = 1,\ldots,L$, the tasks in $S(\ell)$ alone can be (fractionally) scheduled on processors $\mathcal{P}_b$ with makespan at most $\frac{a}{a-1}(1+\delta)^\ell$. Now, using [15, Theorem 2.1], we obtain an integral schedule for the tasks in $S(\ell)$ alone with makespan at most $(\frac{a}{a-1}+1)(1+\delta)^\ell$. By the definition of $S(\ell)$, in this integral schedule, each task $T_{k,j} \in S(\ell)$ has a completion time of at most $a(\frac{a}{a-1}+1)(1+\delta)\bar{C}_{k,j}$. Therefore, if we take the union of these schedules, one after another, in increasing order of $\ell = 1,\ldots,L$, the completion time of each job $j$ is at most $a(\frac{a}{a-1}+1+\frac{1}{\delta})(1+\delta)\bar{C}_{D_j}$. Choosing $a = 3/2$ and $\delta = 1/2$, we obtain that:

**Theorem 1.** *[6] Algorithm* TASKSCHEDULING*($b$) is a $27/2$-approximation for scheduling a set of Map tasks (resp. Reduce tasks) on a set of unrelated processors $\mathcal{P}_\mathcal{M}$ (resp. $\mathcal{P}_\mathcal{R}$), in order to minimize their total weighted completion time.*

**Merging Task Schedules.** Let $\sigma_\mathcal{M}, \sigma_\mathcal{R}$ be two schedules computed by TASKSCHEDULING($b$), for $b = \mathcal{M}$ and $b = \mathcal{R}$, respectively. Let also $C_j^{\sigma_\mathcal{M}} = \max_{T_{k,j}\in\mathcal{M}}\{C_{k,j}\}$ and $C_j^{\sigma_\mathcal{R}} = \max_{T_{k,j}\in\mathcal{R}}\{C_{k,j}\}$ be the completion times of all Map and all Reduce tasks of a job $j \in \mathcal{J}$ within these schedules, respectively. Depending on these completion time values, we assign each job $j \in \mathcal{J}$ a *width* equal to $\omega_j = \max\{C_j^{\sigma_\mathcal{M}}, C_j^{\sigma_\mathcal{R}}\}$. The following algorithm computes a feasible schedule for Map-Reduce scheduling.

*Algorithm* MR. Every time a processor $i \in \mathcal{P}_b$ becomes available, schedule: either the Map task, assigned to $i \in \mathcal{P}_{\mathcal{M}}$ in $\sigma_{\mathcal{M}}$, with the minimum width, or the available (w.r.t. its "release time" $\omega_j$) Reduce task, assigned to $i \in \mathcal{P}_{\mathcal{R}}$ in $\sigma_{\mathcal{R}}$, with the minimum width.

**Theorem 2.** *Algorithm* MR *is a* $54$*-approximation for Map-Reduce scheduling.*

*Proof (Sketch).* By the execution of MR, it is immediate to verify the feasibility of the final schedule. So, it suffices to prove that in such a schedule $\sigma$, all tasks of a job $j \in \mathcal{J}$ are completed by time $2\max\{C_j^{\sigma_{\mathcal{M}}}, C_j^{\sigma_{\mathcal{R}}}\}$. Let $C_j^{\sigma}$, be the completion time of a job $j \in \mathcal{J}$ in $\sigma$. Note that, for each of the Map tasks of $j$, their completion time is upper bounded by $\omega_j$. On the other hand, the completion time of each Reduce task is upper bounded by a quantity equal to $r + \omega_j$, where $r$ is the earliest time when the task is available to be scheduled in $\sigma$. However, $r = C_j^{\sigma_{\mathcal{M}}} \leq \omega_j$ and thus $C_j^{\sigma} \leq 2\omega_j = 2\max\{C_j^{\sigma_{\mathcal{M}}}, C_j^{\sigma_{\mathcal{R}}}\}$. Then, the theorem follows from Theorem 1. □

*Remark.* If the two sets of processors, $\mathcal{P}_{\mathcal{M}}, \mathcal{P}_{\mathcal{R}}$, are not disjoint (or even if they coincide with each other), then setting $\omega_j = C_j^{\sigma_{\mathcal{M}}} + C_j^{\sigma_{\mathcal{R}}}$ and applying a similar analysis, we obtain the same approximation ratio as in Theorem 2.

## 3 The Map-Shuffle-Reduce Scheduling Problem

In practical MapReduce systems, data shuffle represents a significant cost for the key-value pairs with the same key to be transmitted from their Map tasks to the corresponding Reduce task. Motivated by [5], we model this cost by introducing a number of *Shuffle tasks* for each Map task. However, in contrast to [5], where the assignment of Shuffle tasks to processors is fixed, our model distinguishes between two variants: a) Each Shuffle task is scheduled on the same processor as its corresponding Reduce task and b) the Shuffle tasks are scheduled on a different set of processors. For both variants, we present $O(1)$-approximation algorithms.

The number of different keys is usually significantly larger than the number of Reduce processors. Hence, a Reduce task receives all key-value pairs with key in a set of different keys. Allowing the transmission time of some Shuffle tasks to be $0$, we may assume wlog. that all Reduce tasks receive key-value pairs from all Map tasks. We also assume that only a single key-value pair can be transferred to a Reduce processor at any time and moreover, the transmission process cannot be interrupted. Thus, since the key-value pairs allocated to the same Reduce task cannot be transmitted in parallel, we can assume that all key-value pairs from a Map task, assigned to the same Reduce task, can be considered as a single Shuffle task. Hence, the number of Shuffle tasks per Map task equals the number of Reduce tasks. The following summarizes the above discussion.

**Properties:** *i) Each Shuffle task cannot start its execution before the completion of its corresponding Map task.*
*(ii) For every Map task of a job $j$, there are as many Shuffle tasks as $j$'s Reduce tasks. When no key-value pairs are transmitted from a Map task to a Reduce task, the transmission time of the corresponding Shuffle task is equal to* $0$*.*
*(iii) Each Shuffle task is executed non-preemptively.*
*(iv) Shuffle tasks transmitting to the same processor do not overlap.*

Before presenting our results for the Map-Shuffle-Reduce scheduling problem, we introduce some additional notation. For each Map task $T_{k,j} \in \mathcal{M}$ of a job $j \in \mathcal{J}$, we introduce a set of Shuffle tasks $T_{r,k,j}$, $1 \leq r \leq \tau_j$, with $\tau_j$ denoting the number of Reduce tasks of job $j$. We denote by $\mathcal{H}$ the set of Shuffle tasks; note that for each Map task of a job, there is a bijection between its Shuffle tasks and the job's Reduce tasks. Each Shuffle task $T_{r,k,j} \in \mathcal{H}$ is associated with a *transmission time* $t_{r,k,j}$, which is independent of the processor assignment.

**The Shuffle Tasks are Executed on the Reduce Processors.** The key step here is the integration of the Shuffle phase into the Reduce phase. In this direction, we consider a Reduce task $T_{r,j}$ of a job $j$ and let $s_j^r = \{T_{r,k,j} \mid T_{k,j} \in \mathcal{M}\}$ be the set of Shuffle tasks that must be completed before task $T_{r,j}$ starts its execution. The tasks in $s_j^r$ are executed in the same processor as Reduce task $T_{r,j}$. Thus, we obtain that:

**Lemma 1.** *There is an optimal schedule of Shuffle tasks and Reduce tasks on processors of $\mathcal{P}_\mathcal{R}$ such that (i) there are no idle periods, and (ii) all the Shuffle tasks in $s_j^r$ are executed together and are completed exactly before the execution of $T_{r,j}$.*

*Proof.* (i) Let $\sigma$ be a feasible schedule. There are three cases where an idle time can occur: either between the execution of two Shuffle or two Reduce tasks or between a Shuffle and a Reduce task. Since all Shuffle and Reduce tasks are assumed to be available from time zero and there are no precedence constraints among only Shuffle tasks or only Reduce tasks, skipping the idle times in the first two cases can only decrease the objective value of $\sigma$. For the third case, we observe that since the Shuffle tasks precede the corresponding Reduce tasks, skipping the idle intervals can only decrease the completion time of the Reduce tasks. Hence, $\sigma$ can be transformed into a schedule with no idle periods without increasing the its total weighted completion time.
(ii) Let us consider a schedule $\sigma$ that violates the claim and has the last Reduce task $T_{k,j}$ of a job $j$ completed on some processor $i \in \mathcal{P}_\mathcal{R}$. We fix the completion time of $T_{k,j}$ and shift all the Shuffle tasks in $s_j^r$ to execute just before $T_{k,j}$, consecutively and in arbitrary order. Then, the completion time of $j$ remains unchanged, while the completion time of every task preceding $T_{k,j}$ in $\sigma$ may decrease. After a finite number of shifts, we obtain a schedule that satisfies (ii) and has at most the total weighted completion time of $\sigma$. □

Using Lemma 1, we can incorporate the execution of the Shuffle tasks of each job into the execution of the corresponding Reduce tasks. Namely, for each Reduce task $T_{r,j}$ of a job $j$, $1 \leq r \leq \tau_j$, we increase its processing time $p_{i,r,j}$, on each processor $i \in \mathcal{P}_\mathcal{R}$, by a quantity equal to the total transmission time of the Shuffle tasks in $s_j^r$, i.e., equal to $p(s_j^r) = \sum_{T_{r,k,j} \in s_j^r} t_{r,k,j}$. Let $p'_{i,r,j} = p_{i,r,j} + p(s_j^r)$ be the increased processing time for each task $T_{r,j} \in \mathcal{R}$ on processor $i \in \mathcal{P}_\mathcal{R}$, referred to as Shuffle-Reduce task and let $\mathcal{R}_\mathcal{H}$ be the new set of Shuffle-Reduce tasks.

Now, running Algorithm TASKSCHEDULING($b$), for $b \in \{\mathcal{M}, \mathcal{R}_\mathcal{H}\}$, we obtain two $27/2$-approximate schedules, one for the Map tasks and one for the Shuffle-Reduce tasks. Moreover, by considering the same precedence constraints as for the Map and Reduce tasks, we can merge the above schedules by applying Algorithm MR. Despite satisfying Property (i), these dependencies are more general than the precedence constraints among Map and Shuffle tasks of each job, because in order to start the execution

of a Shuffle task, we have to wait for all the Map tasks of a job to finish. However, since the completion time of a job $j$ in the optimal schedule is lower bounded by the completion time in optimal schedules of either the Map or the Shuffle-Reduce tasks, regardless of their precedences, we have that:

**Theorem 3.** *Algorithm* MR *is a* $54$*-approximation for Map-Shuffle-Reduce scheduling.*

**The Shuffle Tasks are Executed on Different Processors.** To deal with this case, we assume that for any processor $i \in \mathcal{P}_{\mathcal{R}}$, there exists an "input" processor which receives data from the Map processors. Therefore, the input processor executes the Shuffle tasks that correspond to the Reduce tasks which have been assigned to processor $i$. We refer to the set of input processors as $\mathcal{P}_{\mathcal{S}}$.

**Lemma 2.** *Consider two optimal schedules* $\sigma$ *and* $\sigma'$ *of Shuffle and Reduce tasks on processors in* $\mathcal{P}_{\mathcal{R}} \cup \mathcal{P}_{\mathcal{S}}$ *and* $\mathcal{P}_{\mathcal{R}}$*, respectively. Let also* $C^{\sigma}_{k,j}, C^{\sigma'}_{k,j}$ *be the completion times of any Reduce task* $T_{k,j}$ *in* $\sigma$ *and* $\sigma'$*, respectively. Then,* $C^{\sigma'}_{k,j} \leq 2C^{\sigma}_{k,j}$*.*

*Proof.* We start with an optimal schedule $\sigma$ on the set $\mathcal{P}_{\mathcal{R}} \cup \mathcal{P}_{\mathcal{S}}$ of processors. We fix a Reduce processor $i^r$, the corresponding input processor $i^s$ and a Reduce task $T_{k,j} \in \mathcal{R}$ of a job $j$. We build the schedule $\sigma'$ on $i^r$ by executing the Reduce tasks in the same order as in $\sigma$ and just before each Reduce task, we execute the corresponding Shuffle task. Let $B(k)$ be the set of Reduce tasks executed on processor $i^r$, before $T_{k,j}$ and let $Sh(k)$ the set of the shuffle tasks that correspond to the Reduce tasks in $B(k) \cup \{T_{k,j}\}$. Then, we have that

$$C^{\sigma'}_{k,j} = \sum_{T_{l,j} \in B(k)} p_{i^r,l,j} + \sum_{T_{q,l,j} \in Sh(k)} t_{q,l,j},$$

which holds because there are no idle intervals in $\sigma'$, by Lemma 1. Moreover, since both $B(k)$ and $Sh(k)$ have to complete before $T_{k,j}$ in $\sigma$, we have that

$$C^{\sigma}_{k,j} \geq \max \left\{ \sum_{T_{l,j} \in B(k)} p_{i^r,l,j}, \sum_{T_{q,l,j} \in Sh(k)} t_{q,l,j} \right\},$$

and therefore $C^{\sigma'}_{k,j} \leq 2C^{\sigma}_{k,j}$. □

Combining Lemma 2 and Theorem 1, we obtain a 27-approximation for scheduling the Shuffle-Reduce tasks. Then, running Algorithm MR, we get the following corollary. Here, the Shuffle tasks form a special third stage in the FFS problem.

**Corollary 1.** *Algorithm* MR *is a* $81$*-approximation for Map-Shuffle-Reduce scheduling, in the general case where the Shuffle tasks run on a separate set of processors.*

## 4 Experimental Evaluation

In this section we experimentally evaluate the performance of Algorithm MR. To deal with data shuffle in our model, in Section 3, we apply Algorithm MR to instances of the

Map-Reduce scheduling problem with increased processing times for the Reduce tasks that take the data transmission time of the Shuffle tasks into account. Thus, to simplify the experimental evaluation, we restrict our attention to instances of Map-Reduce scheduling. We compare the solutions of Algorithm MR, for two different families of random instances, with a lower bound on the optimal cost of Map-Reduce scheduling and with the solutions of a simple and fast scheduling algorithm, called Fast-MR, that we propose below.

To compute a lower bound on the optimal solution of Map-Reduce scheduling, we include in the LP-relaxation LP(b) all the Map and Reduce tasks and also the precedence constraints among them. These dependencies can be captured by the following set of constraints in LP(b):

$$C_{k,j} \geq C_{k,j'} + \sum_{i \in \mathcal{P}_{\mathcal{R}}} \sum_{\ell \in \mathcal{L}} p_{i,k,j} y_{i,k,j,\ell} \qquad \forall j \in \mathcal{J}, T_{k,j} \in \mathcal{R}, T_{k,j'} \in \mathcal{M}.$$

Our Fast-MR algorithm consists of two steps: First it finds a online assignment of tasks to processors and then schedules them using a variant of the well known Weighted Shortest Processing Time first (WSPT) policy.

**Fast-MR.** *Step A*: Apply the online algorithm ASSIGNU, presented in [3] for makespan minimization on unrelated processors: For an arbitrary order of jobs, arriving one-by-one in an arbitrary order, assign each task $T_{k,j}$ of the current job $j$ to the processor $k = \arg\min_{i \in \mathcal{P}_b} \{\lambda^{L_i + p_{i,k,j}} - \lambda^{L_i}\}$, where $L_i$ is the current load of processor $i$ and $\lambda > 1$ an appropriately chosen constant. The tasks of each job are considered one-by-one in an arbitrary order.
*Step B*: Order the tasks assigned to each processor, in Step A, by applying the following version of the WSPT policy:

> For each pair of jobs $j, j' \in \mathcal{J}$, if $(w_j / \sum_{T_{k,j} \in j} p_{k,j}) > (w_{j'} / \sum_{T_{k,j'} \in j'} p_{k,j'})$, then job $j$ precedes $j'$ in the schedule.

When a processor becomes available, schedule the task that is not yet executed, while respecting the precedences among Map and Reduce tasks.

### 4.1 Computational Experiments and Results

We performed the experiments on a machine with 4 packages (Intel(R) Xeon(R) E5-4620 @ 2.20GHz) of 8 cores each (16 threads with hyperthreading) and a total memory of 256 GB. The operating system was a Debian GNU/Linux 6.0. We used Python 2.7 for scripting. The solver used for the linear programs waw Gurobi Optimizer 6.0.

An instance of the problem consists of a $n \times |\{T_{k,j} \in \mathcal{M} \cup \mathcal{R}\}| \times m$ matrix that describes the processing times of the tasks, a vector of size $n$ that describes the job weights, and a precedence graph for the tasks of the same job. We use two disjoint sets of processors, each consisting of 40 Map and 40 Reduce processors. We consider instances from 5 to 50 jobs; each job has 30 map tasks and 10 reduce tasks. Moreover, we fix $\delta = 0.5$ and $a = 1.5$, for the parameters of Algorithm MR. For each of the $n$ jobs, its weight is uniformly distributed in $[1, n]$.

The quality of the solutions in our experiments depend on whether there is any correlation between jobs and processors. Based on [11], in order to experiment with two representative cases, we generate the task processing times in each processor in two different ways, uncorrelated and processor-job correlated, and test experimentally both Algorithm MR and Fast-MR by running 10 different trials for each possible number of jobs. The instances and the code used in our experiments are available at `http://www.corelab.ntua.gr/~opapadig/mrexperiments/`.

*Uncorrelated Input.* The processing times $\{p_{i,k,j}\}_{i\in\mathcal{P}_\mathcal{M}}$ of the Map tasks $T_{k,j} \in \mathcal{M}$ of each job $j$ are selected uniformly at random (u.a.r.) from $[1, 100]$. Similarly to [4], we set the processing times $\{p_{i,k,j}\}_{i\in\mathcal{P}_\mathcal{R}}$ of the Reduce tasks $T_{k,j} \in \mathcal{R}$ to thrice a value selected u.a.r. from $[1, 100]$ plus some "noise" selected u.a.r. from $[1, 10]$.

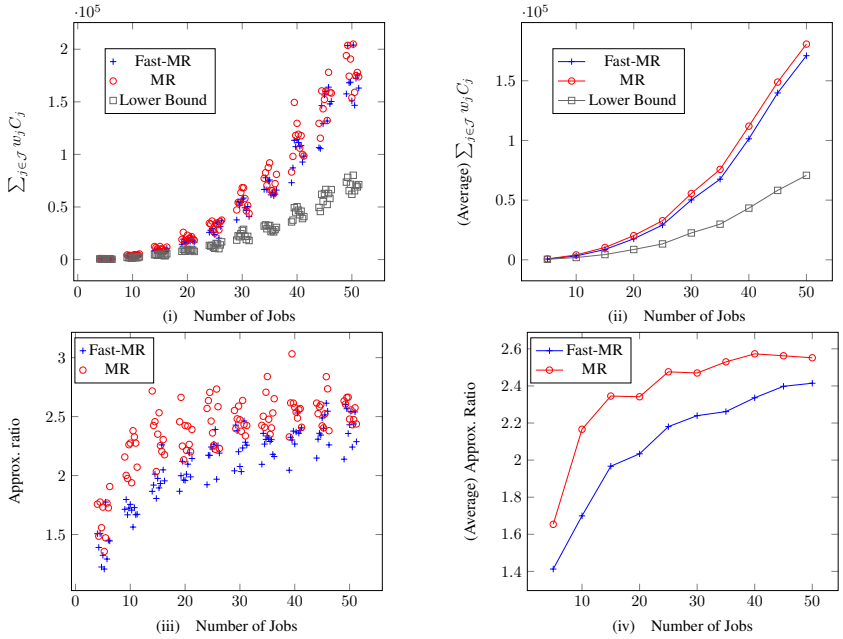


**Fig. 1:** (i)-(ii): The objective values of the solutions found by algorithms Fast-MR and MR and a lower bound on the optimal. (iii)-(iv): The observed approximation ratios of Fast-MR and MR for instances with uncorrelated processing times.

In Fig. 1.(i)-(ii), we observe that Fast-MR performs better than MR in general when the processing times are uncorrelated. For a small number of jobs, Fast-MR gives up to $21\%$ (on average) better solutions. However, as the number of jobs increases, the gap between Fast-MR and MR decreases, e.g., for $n = 45$ and $n = 50$ Fast-MR gives $6\%$ and $5\%$ (on average) better solutions, respectively. In fact, since the processing times are selected u.a.r. from identical uniform distributions, the processors tend to behave as essentially identical, rather than unrelated, which gives a significant practical advantage to Fast-MR, especially for small instances. This holds for both the assignment and the scheduling phase of Fast-MR, since WSPT is also known to perform quite well on identical processors. As for the performance guarantees, as we can see, in Fig. 1.(iii)-(iv),

that the (empirical) approximation ratio of MR ranges from $1.65$ to $2.57$ (on average), while the approximation ratio of Fast-MR ranges from $1.41$ to $2.41$ (on average). These values are far from MR's worst-case approximation guarantee of $54$.

*Processor-Job Correlated Input.* To better capture issues of data locality in our unrelated processors setting, we next focus on instances which use processor and job correlations. In this direction, the processing times $\{p_{i,k,j}\}_{i \in \mathcal{P}_{\mathcal{M}}}$ of the Map tasks $T_{k,j} \in \mathcal{M}$ of each job $j$ are uniformly distributed in $[\alpha_i \beta_j, \alpha_i \beta_j + 10]$, where $\alpha_i$, $\beta_j$ are selected u.a.r. from $[1, 20]$, for each processor $i \in \mathcal{M}$ and each job $j \in \mathcal{J}$ respectively. As before, the processing time of each Reduce task is set to three times a value selected u.a.r. from $[\alpha_i \beta_j, \alpha_i \beta_j + 10]$ plus some "noise" selected u.a.r. from $[1, 10]$.

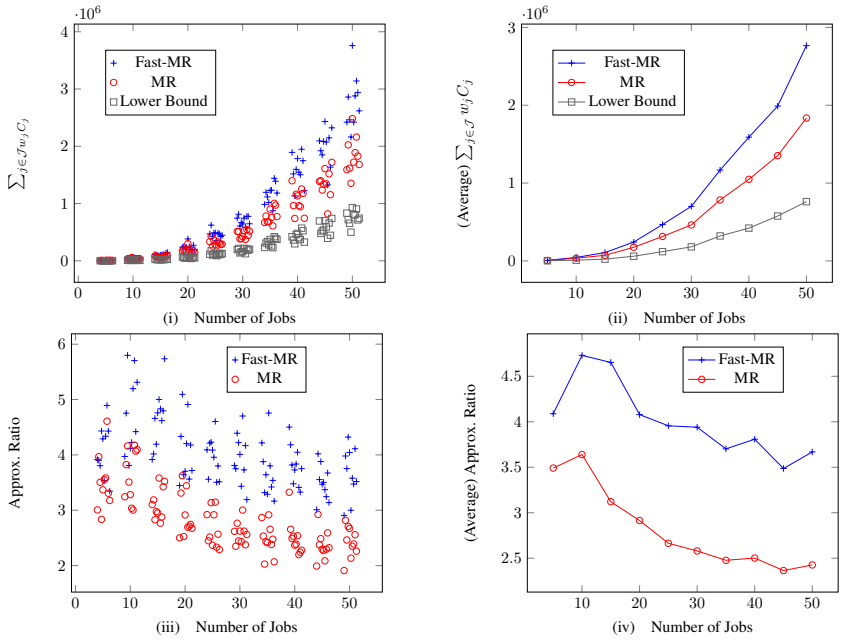


**Fig. 2:** (i)-(ii): The objective values of the solutions found by algorithms Fast-MR and MR and a lower bound on the optimal. (iii)-(iv): The observed approximation ratios of Fast-MR and MR for instances with correlated processing times.

In Fig. 2.(i)-(ii), we observe that Algorithm MR outperforms Fast-MR for any number of jobs. Specifically, MR leads to $11\% - 34\%$ (on average) smaller total weighted completion times than Fast-MR. Due to the processor-job correlation in task processing times, the environment now resembles better that of MapReduce scheduling with unrelated processors and data locality. Then, the more sophisticated assignment and the scheduling procedures of Algorithm MR have a significant advantage over the simple online assignment and WSPT-based scheduling of Fast-MR. In fact, even for a small number of jobs, $n = 5$, Algorithm MR results in up to $11\%$ (on average) better solutions. The (empirical) approximation ratio of MR, in Fig. 2.(iii)-(iv), ranges from $2.42$ to $3.49$ (on average), while, for Fast-MR, the approximation ratio ranges from $3.48$ to $4.73$ (on average). Again, both algorithms are far from MR's worst-case approximation

guarantee of $54$. Furthermore, we observe that the empirical approximation ratio of MR (and also its advantage over Fast-MR) seem to improve as the number of jobs increases and the assignment and scheduling problem becomes more demanding.

## References


1. F.N. Afrati, A. Das Sarma, S. Salihoglu, and J.D. Ullman. Upper and Lower Bounds on the Cost of a MapReduce Computation. *Proceedings of VLDB*, 6(4):277–288, 2013.
2. F.N. Afrati and J.D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1282–1298, 2011.
3. J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling. *Journal of the ACM*, 44(3):486–504, 1997.
4. 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 Proceedings of the 30th International Conference on Computer Communications*, pages 3074–3082, 2011.
5. F. Chen, M. S. Kodialam, and T. V. Lakshman. Joint scheduling of processing and shuffle phases in mapreduce systems. In *IEEE Proceedings of the 31st International Conference on Computer Communications*, pages 1143–1151, 2012.
6. J. R. Correa, M. Skutella, J. Verschae. The power of preemption on unrelated machines and applications to scheduling orders. *Mathematics of Operations Research*, 37(2):379–398, 2012.
7. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 137–150, 2004.
8. M. R. Garey, D.S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.
9. T. Gonzalez and S. Sahni. Flowshop and jobshop schedules: complexity and approximation. *Operations research*, 26(1):36–52, 1978.
10. L. A. Hall, A.S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.
11. A. M. Hariri, and C. N. Potts. Heuristics for scheduling unrelated parallel machines. *Computers and Operations Research*, 18(3):323–331, 1991.
12. M. Mastrolilli, M. Queyranne, A. S. Schulz, O. Svensson, and N. A. Uhan. Minimizing the sum of weighted completion times in a concurrent open shop. *Operations Research Letters*, 38(5):390–395, 2010.
13. B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós. On scheduling in map-reduce and flowshops. In *Proc. of the 23rd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 289–298, 2011.
14. P. Schuurman and G. J. Woeginger. A polynomial time approximation scheme for the two-stage multiprocessor flow shop problem. *Theoretical Computer Science*, 237(1):105–122, 2000.
15. D.B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.
16. D.-J. Yoo and K. M. Sim. A comparative review of job scheduling for mapreduce. In *IEEE Proc. of the International Symposium on Cloud Computing and Intelligece Systems*, pages 353–358, 2011.