

# Scheduling in MapReduce-like Systems for Fast Completion Time

Hyunseok Chang<sup>†</sup>, Murali Kodialam<sup>†</sup>, Ramana Rao Kompella<sup>‡</sup>, T. V. Lakshman<sup>†</sup>, Myungjin Lee<sup>‡</sup>, Sarit Mukherjee<sup>†</sup>  
<sup>†</sup>Bell Labs Alcatel-Lucent, <sup>‡</sup>Purdue University

**Abstract**—Large-scale data processing needs of enterprises today are primarily met with distributed and parallel computing in data centers. MapReduce has emerged as an important programming model for these environments. Since today’s data centers run many MapReduce jobs in parallel, it is important to find a good scheduling algorithm that can optimize the completion times of these jobs. While several recent papers focused on optimizing the scheduler, there exists very little theoretical understanding of the scheduling problem in the context of MapReduce. In this paper, we seek to address this problem by first presenting a simplified abstraction of the MapReduce scheduling problem, and then formulate the scheduling problem as an optimization problem. We devise various online and offline algorithms to arrive at a good ordering of jobs to minimize the overall job completion times. Since optimal solutions are hard to compute (NP-hard), we propose approximation algorithms that work within a factor of 3 of the optimal. Using simulations, we also compare our online algorithm with standard scheduling strategies such as FIFO, Shortest Job First and show that our algorithm consistently outperforms these across different job distributions.

## I. INTRODUCTION

MapReduce [1] has emerged as an important paradigm in many large-scale data processing applications in modern data centers. These applications include search indexing, mining social networks, recommendation services and advertising backends, to name a few. While popularized by Google [1], MapReduce is used by several companies including Microsoft, Yahoo, Facebook in their clusters for many of their applications today.

At a high-level, MapReduce essentially consists of two functions—map and reduce. During the map phase, the entire dataset (that may include millions of Web documents) is partitioned into several smaller chunks which are assigned to an individual nodes (where the chunks reside) for partial computation of results. The results are computed in the form of key-value pairs on the original data set. For example, a simple word counting application may output tuples of the form  $\langle \text{word}, \text{number of occurrences} \rangle$  within the particular chunk. During the reduce phase, each node (that executes the reduce task) will contact all nodes that possess the key-value pairs generated as a result of the initial map phase and aggregate these tuples based on the key. For example, in the word counting application, a reduce task will add up the counts of all the tuples which have the same key (word in this case). Thus, a complex job can be easily partitioned into map and reduce tasks, and can be executed in parallel, with synchronization happening only between the phases.

At any given time, a typical large-scale data center (*e.g.*, owned by Google or Microsoft), may be running many MapReduce jobs, with each job comprising of multiple map and reduce tasks. A centralized *master* orchestrates the assignment and scheduling functions across all jobs in the data center. The assignment and scheduling functions essentially determine *which node* to assign a particular task (either map or reduce) to and *when* to schedule it. The master keeps track of progress of individual tasks (through heartbeat messages sent by the individual nodes) to determine where to assign and when to schedule individual tasks. Prior experience [1] shows that one (or a few for fault tolerance) centralized entity can scalably perform these functions for all the jobs in the entire data center.

The success and widespread acceptance of the MapReduce paradigm has generated a lot of interest in the networking and systems research community to optimize MapReduce, especially the assignment and scheduling functions [2], [3], [4]. While these efforts certainly improve the default MapReduce scheduling and assignment strategies to some extent, they are largely based on practical insights, that are validated with empirical evaluations. To the best of our knowledge, however, there exists very little *theoretical understanding* of the scheduling and assignment problems (henceforth, referred to as just the scheduling problem) in the context of MapReduce—a limitation we seek to address in this paper.

Scheduling by itself is certainly not a new problem; indeed, a long history of foundational work exists in the literature (*e.g.*, [5]). However, considering the importance and relevance of MapReduce to modern data centers, we believe the time is right to create a theoretical framework for optimizing MapReduce. In this paper, we focus mainly on devising an optimal scheduling algorithm—which is only part of the MapReduce system—for tractability and ease of exposition; we note, however, that our simulation framework for evaluating the scheduling algorithms involves the full system. We cast the scheduling problem as a novel optimization problem that focuses on finding an optimal schedule that minimizes the completion time of all jobs in the cluster. Finding an optimal schedule even for the *offline* version of the problem, where job arrivals are known beforehand, turns out to be an NP-hard problem. We therefore develop a 3-approximate algorithm called OFFA for the offline case, and an algorithm called ONA for the online version (a heuristic described in more detail in Section II).

Using simulations, we compare our scheme with other standard scheduling algorithms such as Shortest Job First (SJF), FIFO, Shortest Task First (STF). The key observation we make is that while SJF or STF may seem to work well under specific workloads, no algorithm performs *consistently* well across different workloads. Because our online algorithm is based on an optimization framework, it performs consistently better than other schemes in terms of scheduling jobs for faster completion times, clearly showing the importance of our framework.

Thus, our paper makes the following contributions. 1) We introduce a theoretical framework for optimal scheduling in MapReduce. To the best of our knowledge, no such framework exists in literature. 2) We devise a novel scheduling algorithm that minimizes the job completion times in the cluster with a large number of machines. We formulate an LP for determining the optimal offline algorithm, but the naive formulation leads to an exponential number of constraints. To solve this more efficiently, we derive a 3-approximation algorithm called OFFA. 3) We show that an online version of our algorithm called ONA achieves 30% shorter completion time of all jobs compared to the original FIFO-based scheduling via simulation. Our algorithm exhibits stable performance regardless of job size distributions while simple heuristics such as shortest tasks first (STF) and shortest job first (SJF) have high variance in completion time depending on job size distributions.

## II. THEORETICAL FRAMEWORK FOR SCHEDULING

### A. Preliminaries and Problem Definition

Consider a system comprising of  $m$  processors<sup>1</sup> that has to process  $n$  jobs. We use  $M$  to represent the set of processors and  $J$  to represent the set of jobs that have to be processed. Each job  $j \in J$  comprises of  $n_j$  tasks each of which has to be processed on a pre-specified processor. Let  $M_j \subseteq M$  represent the set of processors required to process job  $j$ . Assume for simplicity, that each processor can process at most one task at a time, and tasks need to be processed in a *non-preemptive manner*. In practice, of course, processors typically run multiple tasks simultaneously and preempt each other (when a task is blocking on some I/O, for instance), but we make this assumption mainly to keep task processing times independent of each other.

We assume that there are no precedence relationships between different tasks belonging to a given job or between different jobs. In reality, however, reduce tasks are scheduled<sup>2</sup> after map tasks are done. We assume the lack of precedence for theoretical tractability, but we note that in our simulations, we model this precedence accurately. Each job  $j$  is assumed to be available for processing at time  $r_j$ , called the release time or arrival time for job  $j$ . Finally, since in many data centers, one could rank jobs in some order of importance—for example, search indexing jobs may be more important than data

analysis jobs—we assume each job also has a weight  $w_j$  that specifies its importance.

We need two more definitions before we describe the objective of the scheduling function. We define the *finish time* of job  $j$  on processor  $p \in M_j$ , denoted by  $f_{jp}$ , to be the time at which the task of job  $j$  is completed on processor  $p$ . We also define the *completion time* of a job,  $C_j$  as the time at which all tasks belonging to a job  $j$  are finished. Therefore the completion time of job  $j$ , is  $C_j = \max_{p \in M_j} f_{jp}$ . If there are multiple tasks assigned to the same processor  $p$ , as can happen in real MapReduce systems, we consider the union of all the tasks as a larger task with its finish time the maximum finish time among its constituents. Since we are mainly concerned with the completion time of a job, this ‘bundling’ of tasks will not cause any problem in the model. We use  $J_p$  to denote the set of jobs that have a task assigned to processor  $p$ . Therefore  $j \in J_p$  if and only if  $p \in M_j$ .

The *objective* of the scheduler is to determine the ordering of the tasks on each processor in order to minimize the weighted sum of the job completion times, *i.e.*,  $\sum_{j \in J} w_j C_j$ . In the offline scheduling problem, all job arrival times,  $r_j$  are known in advance. The true scheduling problem, however, is the *online problem* where jobs are released over time and the job release times are not known to the scheduler in advance. Also, while we assume processing times are known for simplicity, in reality, they are usually estimated, and therefore may not be known accurately; we therefore account for estimation errors in our model.

**Problem Complexity** Even on a single processor, the problem of scheduling jobs with release times to minimize the weighted sum of completion times is NP-hard [6]. Therefore our problem is also NP-hard. We derive a 3-approximation algorithm for our problem based on solving a linear program (LP) relaxation. On a single processor, if all the arrival times are zero, then scheduling jobs in decreasing order of the weight to processing time ratio (Smiths Rule) minimizes the weighted completion time. This is not true for our problem. The complexity of the problem is open. However, we derive a 2-approximation algorithm for this problem.

### B. Algorithmic Approach

We use the following three-step algorithmic approach to solve the scheduling problem:

- 1) We derive an LP that gives a lower bound on the optimal solution. The LP solution may not give a feasible schedule.
- 2) We use the LP solution to derive a feasible schedule that is within a factor of 3 of the optimal solution.
- 3) We also show a slightly tighter approximation (2-approximation) for the special case when all jobs are available at time  $t = 0$ , *i.e.*,  $r_j = 0$  for all jobs. This special case becomes the basis for our online scheduling algorithm.

<sup>1</sup>In multicore systems, we consider each core as a separate processor.

<sup>2</sup>In some cases, the scheduler may schedule the reduce tasks earlier, but they can only start processing after the map tasks are finished.

### C. LP-based Lower Bound

Given the processing times, release times, weights and the allocation of tasks to processors, we now derive a linear programming relaxation of the scheduling problem that gives a lower bound on the optimal solution value. The constraints of the LP are necessary conditions that the completion time of jobs has to satisfy. These conditions can be viewed as multiprocessor extension to the single processor scheduling polyhedron derived in [5].

*Theorem 2.1:* Let  $t_{jp}$  denote the processing time of job  $j$  on processor  $p$ , and  $J_p$  denote the set of jobs on processor  $p$ . Then the following LP (LP\_OPT\_SCHED) provides a lower bound on the minimum weighted completion time schedule:

$$Z^{LP} = \min \sum_{j \in J} w_j C_j$$

$$\sum_{j \in S} t_{jp} C_j \geq f(S, p) \quad \forall S \subseteq J_p \quad \forall p \quad (1)$$

$$C_j \geq r_j + h_j \quad \forall j \quad (2)$$

where

$$f(S, p) = \frac{1}{2} \left[ \sum_{j \in S} t_{jp}^2 + \left( \sum_{j \in S} t_{jp} \right)^2 \right] \text{ and } h_j = \max_{p \in M_j} t_{jp}.$$

*Proof:* We outline the proof for completeness. For a more detailed explanation see the proof of the single processor problem in [5]. Fix a processor  $p$  and consider the set of jobs  $J_p$  that are assigned to processor  $p$ . Consider a subset  $S \subset J_p$  and let  $S = \{1, 2, \dots, k\}$ . Let  $f_{jp}$  denote the finish time of job  $j$  on processor  $p \in M_j$ . Assume that the jobs in  $S$  are processed before any job in  $J_p \setminus S$ . Assume that the jobs are processed in the order  $1, 2, \dots, k$ . Since we are deriving a lower bound, we ignore  $r_j$  and in this case  $f_{jp} = \sum_{i=1}^j t_{ip}$  and

$$\sum_{j \in S} t_{jp} f_{jp} = \sum_{j=1}^k t_{jp} \left( \sum_{i=1}^j t_{ip} \right) = f(S, p).$$

Note that  $f(S, p)$  is symmetric in  $t_{jp}$  and therefore its value is *independent* of the order in which the jobs in set  $S$  are processed. Moreover, it is easy to show that the  $\sum_{j \in S} t_{jp} f_{jp}$  only increases if some job in  $J_p \setminus S$  is processed along with jobs in  $S$ . This holds for any subset  $S$  of  $J_p$ . We now use the fact that  $C_j \geq f_{jp}$  to write  $\sum_{j \in S} t_{jp} C_j \geq \sum_{j \in S} t_{jp} f_{jp}$  thus giving the constraints in (1). The constraints in (2) just state that the completion time of job  $j$  cannot be less than the sum of the arrival time of the job and the maximum processing time of the job on any processor. ■

Note that in Equation (1) there are an exponential number of constraints for each processor. The polyhedron for each processor is a polymatroid. In our case, the job completion times lie in the intersection of  $m$  polymatroids one belonging to each processor. We should point out that the constraints in the LP only represent necessary conditions [5], and therefore,  $Z^{LP}$ , the optimal solution to the linear program, only represents a lower bound of the optimal solution. We illustrate this

by the following example: Consider a two job, two processor problem. The processing times are  $t_{11} = 1, t_{12} = 2, t_{21} = 2, t_{22} = 1$ . Let  $w_1 = w_2 = 1$  and  $r_1 = r_2 = 0$ . In this case the linear programming problem is

$$\begin{aligned} \min C_1 + C_2 \\ C_1 &\geq 1 & 2C_1 &\geq 4 \\ 2C_2 &\geq 4 & C_2 &\geq 1 \\ C_1 + 2C_2 &\geq 7 & 2C_1 + C_2 &\geq 7 \end{aligned}$$

The three constraints on the left are for processor 1 and the three constraints on the right are for processor 2. The optimal solution to the LP is  $C_1 = C_2 = \frac{7}{3}$  giving  $Z^{LP} = \frac{14}{3} = 4.67$ . An optimal schedule is to schedule job 1 on both machines first, followed by job 2. The completion time of job 1 is 2 and the completion time of job 2 is 3 giving a total completion time of 5. Since the LP does not give a feasible solution to the problem, we still need an algorithm that gives a feasible ordering of jobs on different processors, for which we will obtain a 3-approximation algorithm later in Section II-E.

Even solving the LP in polynomial time is difficult, since it has an exponential number of constraints. However, we can solve the LP approximately to any desired level of accuracy using a primal-dual fully polynomial time approximation scheme (FPTAS), which we explain next.

### D. Solving the Linear Program LP\_OPT\_SCHED

One approach to solve LP is to use the ellipsoid algorithm with a separation oracle. Though this will result in a polynomial time algorithm, it is not a practical approach due to the high complexity associated. Thus, we use the primal-dual based FPTAS to solve the LP. The running time of the algorithm increases with the accuracy needed, but the algorithm itself is quite simple to implement.

In order to keep the exposition simple, we outline the solution to a special case of the problem, when all  $r_j = 0$ , that corresponds to the case when all jobs are released to the scheduler at the beginning itself. Note that this problem too has an exponential number of constraints. We call this problem LP\_BULK\_ARRIVAL that is described next; we will outline the changes needed to solve LP\_OPT\_SCHED after solving this problem.

$$\begin{aligned} \min \sum_{j \in J} w_j C_j \\ \sum_{j \in S} t_{jp} C_j \geq f(S, p) \quad \forall S \subseteq J_p \quad \forall p \end{aligned} \quad (3)$$

Instead of solving the above LP directly, we use the primal-dual approach to solve the dual problem which is easier. We associate a dual variable of  $\pi_p^S$  with equation(3). The dual then is

$$\begin{aligned} \max \sum_p \sum_{S \subseteq J_p} f(S, p) \pi_p^S \\ \sum_p t_{jp} \sum_{S \subseteq J_p: S \ni j} \pi_p^S \leq w_j \quad \forall j. \end{aligned}$$

Algorithm PRIMAL\_DUAL

- 1) Initialize  $C_j = \frac{\delta}{w_j}$  for all jobs  $j$ .
- 2) Find the processor  $\bar{p}$  and subset of jobs  $\bar{S} \subseteq J_p$  that minimizes Equation (4).
- 3) Find the job  $\bar{j} \in \bar{S}$  that minimizes  $\frac{w_j}{p_{j\bar{p}}}$ .
- 4) Set  $\pi_{\bar{p}}^{\bar{S}} \leftarrow \pi_{\bar{p}}^{\bar{S}} + \frac{w_{\bar{j}}}{p_{\bar{j}\bar{p}}}$ .
- 5) Set  $C_j \leftarrow C_j + \epsilon \left( 1 + \frac{w_j/p_{j\bar{p}}}{w_{\bar{j}}/p_{\bar{j}\bar{p}}} \right)$  for all jobs  $j$ .

Fig. 1. Description of algorithm PRIMAL\_DUAL

Here,  $\pi_p^S$  are the dual variables and  $C_j$  are the primal variables.

The objective now is to devise a fast combinatorial algorithm that solves the dual (maximization) problem to any pre-specified degree of accuracy. Given any  $\epsilon > 0$ , we outline a combinatorial approach to solve the maximization linear program within a factor of  $(1 - \epsilon)^2$  of the optimal solution. This primal-dual approach works since the dual is a packing type linear program [7].

The primal-dual algorithm starts by initializing completion time  $C_j = \delta/w_j$  for all jobs  $j$ , where  $\delta$  is a parameter that is computed based on  $\epsilon$ . (We give the appropriate value of  $\delta$  in Theorem 2.2.) In each iteration of the primal dual algorithm, we first identify the processor subset combination  $(\bar{p}, \bar{S})$  that minimizes

$$\frac{\sum_{j \in S} t_{jp} C_j}{f(S, p)} \quad (4)$$

for the current values of  $C_j$ . Note that this processor-subset combination represents a column in the linear program. This is equivalent to picking the entering variable in standard simplex algorithm. The efficiency of the primal-dual approach hinges crucially on whether we can pick this entering variable efficiently. In general, this problem may not be easy. The structure of the scheduling polyhedron makes this problem easy to solve. We show how to solve this problem efficiently in a little while, but for now, assume this can be solved. Next, we find the blocking row. The blocking row is the job that minimizes

$$\frac{w_j}{p_{j\bar{p}}}.$$

Then the  $C_j$  values are updated and the process is repeated. We give the outline of the algorithm in Figure 1. The proof of the following theorem is a straightforward adaptation of the proof of the primal-dual scheme in [7].

**Theorem 2.2:** Given an  $\epsilon > 0$ , setting  $\delta = (1 + \epsilon) \left( (1 + \epsilon)n \right)^{\frac{1}{\epsilon}}$ , the algorithm PRIMAL\_DUAL computes  $(1 - \epsilon)^2$  approximate optimal solution to the linear programming problem in at most  $n^{\frac{1}{\epsilon}} \log_{1+\epsilon} n$  iterations, where  $n$  is the number of jobs in the system. The running time for each iteration is the time taken to solve Step 2 of the algorithm.

**Solving the Minimum Column Problem.** A crucial step in the solution of the linear programming problem is to determine

the processor-subset combination that minimizes Equation (4). Note that the completion time values  $C_j$  are known. On any given machine there are however an exponential number of subsets that have to be checked. Therefore a straightforward brute force approach will not work. We show that the structure of the scheduling polyhedron can be exploited to develop a simple polynomial time algorithm to solve this problem. The basic idea is to show that on any processor at most  $n$  sets have to be checked. (Recall that  $n$  is the number of jobs in the system.)

**Theorem 2.3:** Renumber the jobs  $J_p$  on processor  $p$  such that  $C_1 \leq C_2 \leq \dots \leq C_k$  where  $|J_p| = k$ . Then Equation (4) is minimized by some set  $S$  of the form  $\{1, 2, \dots, m\}$  for some  $m = 2, 3, \dots, k$ .

*Proof:* Assume that we fix a processor  $p$  and we want to determine the subset  $S$  that minimizes the expression in Equation (4). Assume that  $S$  is the set minimizes Equation (4) for processor  $p$ . Find some job  $k \in S$ , and let us evaluate what happens to Equation (4) when we evaluate it for  $S \setminus k$ . Let  $g(S, p) = \sum_{j \in S} t_{jp} C_j$ .

$$\frac{g(S \setminus k, p)}{f(S \setminus k, p)} = \frac{g(S, p) - t_{kp} C_k}{f(S, p) - \left( \sum_{j \in S} t_{jp} \right) t_{kp}}$$

Since  $S$  minimizes Equation (4), after doing some algebra, we get

$$C_k \leq \frac{g(S, p)p(S, p)}{f(S, p)}$$

where  $P(S, p) = \sum_{j \in S} t_{jp}$ . Now consider some  $k \in J_p \setminus S$ , then

$$\frac{g(S \cup k, p)}{f(S \cup k, p)} = \frac{g(S, p) + t_{kp} C_k}{f(S, p) + t_{kp}^2 + \left( \sum_{j \in S} t_{jp} \right) t_{kp}}$$

Again, since  $S$  minimizes Equation (4), we can show that

$$C_k \geq \frac{g(S, p)p(S, p)}{f(S, p)} + \frac{t_{kp}g(S, p)}{f(S, p)}.$$

From the above note that  $k \in S$  if and only if

$$C_k \leq \frac{g(S, p)p(S, p)}{f(S, p)}$$

This immediately implies that if we consider two jobs  $j, k \in J_p$ , with  $C_j \leq C_k$  then  $j \in S$  if  $k \in S$  and the result follows. ■

Since each job can potentially be assigned to each processor, the running time of the algorithm is  $O(nm)$ . The primal-dual algorithm to solve LP\_OPT\_SCHED is very similar to the primal-dual algorithm outlined above. The only change is that there are  $n$  additional dual variables corresponding to the release time constraint for each job. Since there are only  $n$  additional variables, these are updated separately. The bottleneck operation is still finding the processor-set combination and therefore the running time of the algorithm remains the same as in Theorem 2.2.

### Algorithm *OFFA*

- 1) Solve LP\_OPT\_SCHED and let  $C_j^{LP}$  denote the completion time of job  $j$ . On each processor  $p$ , the jobs in  $J_p$  are processed in the order output by the LP.
- 2) Given a job  $j \in J_p$ , let

$$P(j, p) = \{k : k \in J_p, C_k^{LP} \leq C_j^{LP}\}$$

denote the set of jobs on  $p$  whose LP completion time is lower than  $j$ . These are jobs that will be scheduled before job  $j$  on processor  $p$ .

- 3) A job is started as soon as the previous job is complete or the job becomes available whichever is later. The finish time of job  $j$  on machine  $p$  denoted by  $f_{jp}^{OFFA}$  is

$$f_{jp}^{OFFA} = \max \left\{ r_j, \max_{k \in P(j,p)} f_{kp}^{OFFA} \right\} + t_{jp}.$$

- 4) The completion time of job  $j$

$$C_j^{OFFA} = \max_{p \in M_j} f_{jp}^{OFFA}.$$

Fig. 2. Description of algorithm *OFFA*

### E. Approximation Algorithm

In this section, we outline how to use the solution to LP\_OPT\_SCHED in order to design a 3-approximation algorithm for the offline scheduling problem. We refer to this algorithm as *OFFA*. This algorithm can be viewed as multiprocessor generalization of the approximation algorithm in [6]. In the rest of this section we use the superscript *LP* to denote the LP solution and *OFFA* to denote the approximation algorithm. Let  $C_j^{LP}$  denote the job completion times corresponding to the optimal LP solution. The algorithm *OFFA* to get the optimal ordering of jobs on processors is shown in Figure 2. We now derive the performance guarantee provided by algorithm *OFFA*.

*Theorem 2.4:* Let  $Z^{LP}$  and  $Z^{OFFA}$  denote the objective function value of LP\_OPT\_SCHED and the sum weighted completion time from algorithm *OFFA* respectively. Then

$$Z^{OFFA} \leq 3 Z^{LP}.$$

*Proof:* Let  $f_{jp}^{OFFA}$  denote the finish time of job  $j$  on processor  $p$  and

$$P(j, p) = \{k : k \in J_p, C_k^{LP} < C_j^{LP}\} \cup \{j\}.$$

Note that  $f_{jp}^{OFFA} \leq \max_{k \in P(j,p)} r_k + \sum_{k \in P(j,p)} t_{kp}$ . From Equation (2), note that  $r_k \leq C_k^{LP}$  and  $C_k^{LP} \leq C_j^{LP}$  from the definition of  $P(j, p)$ . Therefore,

$$f_{jp}^{OFFA} \leq C_j^{LP} + \sum_{k \in P(j,p)} t_{kp}. \quad (5)$$

Applying Equation (1) to the set  $S = P(j, p)$  on processor  $p$ , we get

$$\sum_{k \in P(j,p)} t_{kp} C_k^{LP} \geq f(P(j, p), p) \geq \frac{1}{2} \left( \sum_{k \in P(j,p)} t_{kp} \right)^2.$$

Using the fact that  $C_j^{LP} \geq C_k^{LP}$  for all  $k \in P(j, p)$ , we can write

$$C_j^{LP} \sum_{k \in P(j,p)} t_{kp} \geq \frac{1}{2} \left( \sum_{k \in P(j,p)} t_{kp} \right)^2.$$

Therefore  $\sum_{k \in P(j,p)} t_{kp} \leq 2C_j^{LP}$ . Applying these to Equation(5), gives

$$f_{jp}^{OFFA} \leq 3 C_j^{LP}.$$

This result holds for all  $p \in M_j$ . Since  $C_k^{OFFA} = \max_{p \in M_j} f_{jp}^{OFFA}$ , we have  $C_k^{OFFA} \leq 3 C_k^{LP} \forall k$  and this implies that the solution given by algorithm *OFFA* is less than three times the optimal solution. ■

### F. Stronger Guarantees

For the special case when all jobs are available initially, *i.e.*, for the LP\_BULK\_ARRIVAL problem, the approach before gives a stronger performance guarantee. We outline this result next. If all the jobs are available initially, then the LP corresponding to this problem is

### LP\_BULK\_ARRIVAL

$$Z_B^{LP} = \min \sum_j w_j C_j$$

$$\sum_{j \in S} t_{jp} C_j \geq f(S, p) \quad \forall S \subseteq J_p \quad \forall p$$

This LP provides a lower bound and the LP solution is used to generate an approximation algorithm for the offline problem where all jobs are available at time zero. We refer to this algorithm as *OFFB*. Approximation algorithm *OFFB* to give a solution to this problem operates exactly like algorithm *OFFA*. Jobs are processed on processor  $p$  in the order of the LP\_BULK\_ARRIVAL completion times. Since there are no job arrival times,

$$f_{jp}^{OFFB} = t_{jp} + \sum_{k \in P(j,p)} t_{kp}.$$

*Theorem 2.5:* Let  $Z_B^{LP}$  and  $Z^{OFFB}$  denote the objective function value of LP\_BULK\_ARRIVAL and the sum weighted completion time from algorithm *OFFB* respectively. Then

$$Z^{OFFB} \leq 2 Z_B^{LP}.$$

*Proof:* As stated earlier, algorithm *OFFB* processes jobs in the order of the LP completion times

$$f_{jp}^{OFFB} = \sum_{k \in P(j,p)} t_{kp}. \quad (6)$$

Define the set  $P(j, p)$  as in the proof of Theorem 2.4. Applying Equation (1) to the set  $S = P(j, p)$  on processor  $p$ , and going through the same steps as in Theorem 2.4, we get  $\sum_{k \in P(j, p)} t_{kp} \leq 2 C_j^{LP}$ . Applying these to Equation(6), gives  $f_{jp}^{OFFB} \leq 2 C_j^{LP}$ . This result holds for all  $p \in M_j$ . Since  $C_k^{OFFB} = \max_{p \in M_j} f_{jp}^{OFFB}$ , we have  $C_k^{OFFB} \leq 2 C_k^{LP}$  for all  $k$  and this implies that the solution given by algorithm *OFFB* is less than two times the optimal solution. ■

### G. Online Algorithm

Thus far, we have assumed that the release time (arrival time) of all the jobs are known in advance. This is clearly not true in practice where the jobs arrive one at a time, and the arrival times are not known in advance. In the single processor case there are 2-competitive algorithms for scheduling jobs online. These algorithms use the fact that scheduling the highest weight to processing time ratio job minimizes weighted completion time when all jobs are available at time zero. This is not true for our problem. The approach that we use is the following: When a new job arrives into the system, we take all the jobs currently in the system along with the new job and run *LP\_BULK\_ARRIVAL* with these jobs. We then use the result in Theorem 2.5 to derive the optimal schedule. This schedule is implemented until the arrival of the next job. We refer to this algorithm in the evaluation section as *ONA*. As a refinement, one could just gather a group of arrivals and run the LP periodically. Since we use a fast combinatorial algorithm to solve the LP, solving the LP itself is not a big overhead. This is especially true since we can easily adjust the value of approximation  $\epsilon$  to run the LP faster if needed. We show in the next section that this approach works extremely well and is very robust to varying job characteristics.

## III. EVALUATION

In this section, we present a simulation study of our on-/off-line algorithms. We focus on two aspects. First, we show our offline algorithms—*OFFA* and *OFFB*—work within theoretical bounds that we proved in Section II. Second, we evaluate the efficacy of our online algorithm called *ONA* compared to other candidate approaches. We begin our discussion by introducing scheduling strategies that we used in our evaluation.

### A. Scheduling Strategies

We perform simulations with the following three scheduling strategies along with our offline and online algorithms.

**Shortest Job First (SJF).** We assign tasks whose total processing time (*i.e.*, job size) is the shortest to a cluster first. This strategy appears to be a reasonable greedy algorithm minimizing the completion time, so we take it into account.

**Shortest Task First (STF).** The difference between SJF and this strategy is that STF only uses information that is locally available in determining the scheduling: the sizes of individual tasks submitted to a machine.

**First-In First-Out (FIFO).** Tasks are served in the order of arrivals without introducing any reordering of jobs. This strategy is our basis of comparison throughout this section.

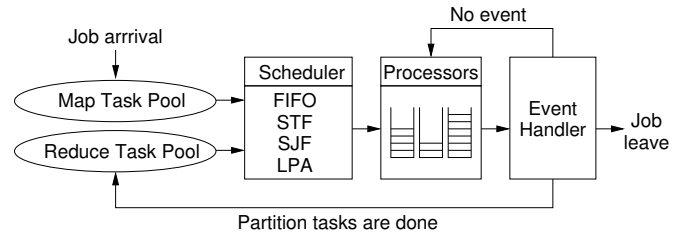


Fig. 3. Structure of simulator.

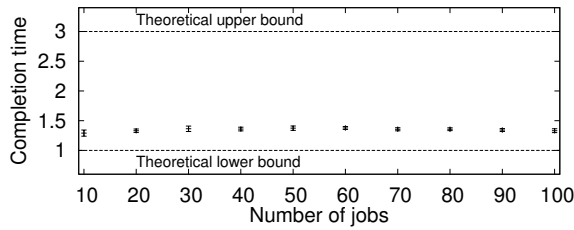
### B. Simulation Setting

While open-source versions of MapReduce exist (*e.g.*, Hadoop), the workloads unfortunately are not publicly available. Most papers (*e.g.*, [4]) use proprietary industry workloads that are not publicly accessible. Thus, we resort to a simulation framework that essentially abstracts (and allows some configuration of) the key characteristics of MapReduce jobs. We also use a synthetic workload generation model that produces a sequence of jobs with different number of tasks and different processing times. We pass this workload information to our scheduler which then emulates different scheduling algorithms. We note that since our results are based on simulations, the exact gains are not to be interpreted literally. Instead, the key takeaway is the fact that our algorithms *consistently* outperform the rest, but by how much depends on the particular workloads.

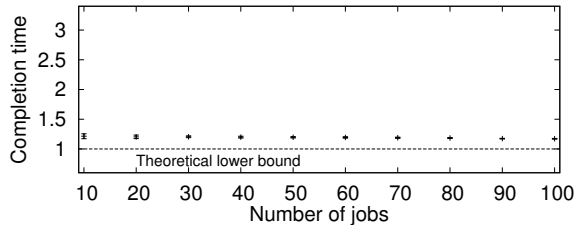
One assumption in our simulation setting is that the processing time of any task is known to the scheduler *a priori*. While this is not true in practice, one can compute an estimate based on the size of the input files, and the types of applications (*e.g.*, sort, database join, page rank, and so on). Since today’s systems produce job logs in significant detail, statistical distribution in processing times of individual tasks may be typically obtained; thus approximation of estimation of job processing time is typically feasible. While we leave the detailed modeling of these processing times as part of future work, we evaluate how sensitive our scheduling strategies are to inaccurate estimation.

Figure 3 shows the overall structure of our simulator. The main parameters for our simulator are global settings such as number of processors. Other parameters such as job release times, number of tasks per job, processing time per task, are drawn from a random distribution. Job arrival is modeled as a Bernoulli random variable with probability 1/2 every unit of time. We assume assignment of tasks has already been made, and that processing times are *independent* of the processor. Each job consists of  $m$  Map tasks and  $n$  Reduce tasks. For each Map task, processing times are uniformly distributed in  $[1, 10]$  units, *i.e.*,  $U[1, 10]$ . Since Reduce tasks typically take longer time than Map to finish, we assign the processing time as  $3 \times T_m/n$ , where  $T_m$  is the total processing time of all Map tasks. We introduce a small amount of randomness by adding a random number of time units between 1 and 10.

We consider two types of job distributions: (1) In the *random distribution* model, we pick each job’s size independently,



(a) Dynamic model with job arrivals using OFFA



(b) Dynamic model with job arrivals using ONA

Fig. 4. Comparison of different algorithms with their theoretical lower bounds of completion time.

with  $m$  drawn from  $U[1, 100]$  and  $n$  drawn from  $U[1, m]$ , the number of Map and Reduce tasks. (2) In the *skewed distribution*, if two jobs are released at the same time, we assign a larger number of tasks for the latter job (by a factor 10), but each task’s processing time is adjusted such that the total job running time (sum of all tasks’ processing times) is similar. We ensure that the former job has a total running time that is smaller than the latter by a small percentage (we use 1% in our simulations). We do this to make this workload unfavorable toward shortest job first, which greedily assigns the first job because it is shorter than the other.

Once the jobs arrive, the scheduler decides which tasks should be executed at currently available machines depending on selected strategies. Individual processors basically make progress on tasks in increments of a unit time. An event handler checks if either all Map tasks or all Reduce tasks for a particular job are completed. If neither of these completions happens, the event handler executes one more unit of work. If all Map tasks of a job are done, then the scheduler is invoked to schedule the Reduce tasks. Once all Reduce tasks for a particular job are completed, the job is finished.

### C. Validating Theoretical Model

We first evaluate how close our approximation algorithm is to the optimal. We then investigate the influence of approximation ratio which is the ratio of dual value to weighted primal value.

**Comparison with lower bound.** In theory, we proved that our algorithm represents a 2-approximation of the theoretical lower bound, when all jobs are available to begin with (static case), *i.e.*,  $r_j = 0$  for all jobs, and a 3-approximation otherwise (dynamic case). Since job arrival times are unknown in practice, our heuristic, ONA, is run at every time new jobs come. We now empirically determine how close our algorithm

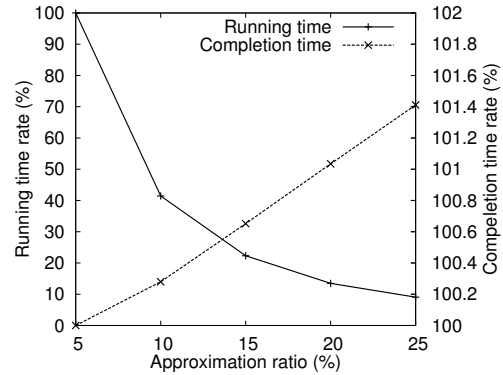


Fig. 5. Impact of different approximation error bound.

is to the lower bound. In this experiment, we vary the number of jobs from 10 to 100. To simulate the dynamic case with job arrivals, we start with an initial set of ten jobs after which jobs arrive until the desired number of jobs have arrived.

The results for the static case (figures omitted for brevity) show that our algorithm’s results represent only a 20% increase in completion time, compared to theoretical lower bound. This implies that empirically, the output of our algorithm is much closer to the lower bound. We observed very little difference even when we increased the number of jobs from 10 to 100. In the dynamic case with job arrivals (shown in Figure 4(a)), we observe a 40% increase in completion time compared with the theoretical lower bound. While this increase is clearly higher than in the static case, recall that the bound we have proved is a 3-approximation for the dynamic case. Similar to the previous case, we observe very little impact on this increase as we vary the number of jobs.

We show the results of our online heuristic, ONA, with dynamic model in Figure 4(b). We observe approximately 25% increase in completion time, compared to theoretical lower bound. Surprisingly, ONA shows shorter completion time than OFFA with global view. Note that OFFA is an 3-approximation algorithm, and in dynamic case, we only run it once since we have global job arrival information. ONA, on the contrary, is based on the 2-approximation algorithm, and it keeps correcting the scheduling order of jobs by running our algorithm whenever jobs come. Thus, ONA appears to perform better than OFFA with global view even though the output of ONA is only based on partial view of the jobs.

**Impact of approximation ratio.** We want to study the trade-off between running time of our algorithm and the completion time of jobs, as we vary the approximation ratio. To this end, we simulate the scenario where hundred jobs are submitted at the same time (static case). We vary approximation ratio from 5% to 25%, and we plot the average of 10 different runs. Figure 5 shows the simulation results normalized to the 5% approximation ratio results (for both algorithm running time and job completion times). In the figure, we can observe that at 10% approximation ratio, running time of the algorithm decreases to almost 40% of the running time at 5% approxima-

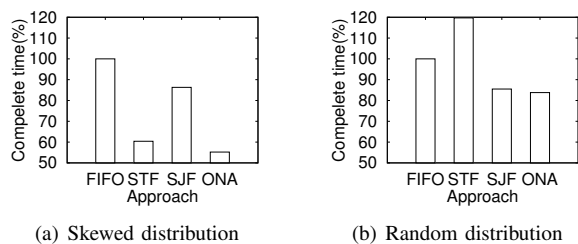


Fig. 6. Static scheduling with no task arrivals.

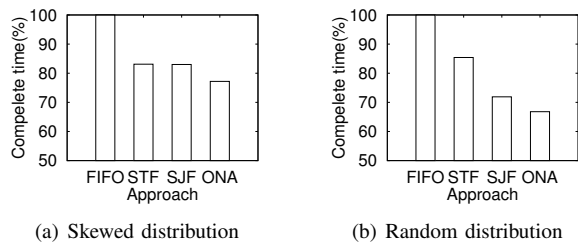


Fig. 7. Dynamic scheduling with Reduce task arrivals.

tion ratio. Further relaxation of the approximation ratio does not yield significant benefits in reducing the algorithm running time. On the other hand, the increase in completion times is relatively slow; even at 25% approximation ratio, there is only 1.5% relative increase compared to the base case.

#### D. Comparison with Other Solutions

**Static scheduling with no task arrivals.** We first compare different scheduling policies for the static case with no job arrivals. From Figure 6, we can see that ONA works best among all other strategies, completing all jobs within about 55% and 84% completion time of FIFO strategy. Another observation is that, while STF finishes jobs shorter than SJF under skewed distribution model, SJF outperforms STF in the random distribution model. In the skewed model, it turns out that SJF first selects jobs with a relatively small number of tasks with large processing time. As a result, jobs with a large number of tasks with small processing time are scheduled later, and eventually increases the total completion time of all jobs. On the contrary, STF first runs the shortest tasks, thus quickly finishing jobs with large number of Map tasks and short processing time of each task, and thus achieves better performance than SJF.

In the random distribution model, STF performs even worse than FIFO because local optimal selection of tasks in STF often hinders tasks belonging to the same job from running simultaneously in different machines while FIFO tries to schedule tasks in a job at the same time. SJF works better than FIFO, but the completion time of SJF is still around 5% higher than that of ONA.

**Dynamic scheduling with Reduce task arrivals.** We now consider the case with job arrival times. Initially, there are ten jobs at a job scheduler, and job arrival process (Bernoulli) is run until one hundred jobs are finished. The results shown in Figure 7 illustrate that ONA performs best among all other

scheduling strategies, followed by SJF. The differences in completion time between ONA and SJF are about 7% and 5% in skewed and random distribution models, respectively. Even though the performance difference between them may look marginal, it may translate into an extra 18–25 days when jobs are run throughout a year. Unlike static scheduling scenarios, SJF always finishes all jobs earlier than STF because new job arrivals mitigate the influence of skewed distribution model in a way that SJF may choose the new job having shorter job size with many tasks, which similarly let SJF work like STF. **Waiting time.** We define the waiting time of a job to be the time interval between the submission and completion times. We found that ONA achieves less than 68% waiting time by FIFO at median, whereas SJF and STF have roughly less than 86% waiting time by FIFO at median as shown in Figure 8(a). Similarly, while SJF finishes 64% of all jobs earlier than FIFO does and STF does 58% of all jobs, about 70% of all jobs scheduled by ONA are finished earlier than ones by FIFO. The figure also shows the obvious trade-off of these scheduling approaches compared to FIFO scheme; some jobs scheduled by schemes other than FIFO experience longer waiting time compared to the case where they are scheduled by FIFO.

**Impact on long jobs.** We only compare ONA with SJF to understand the influence of these scheduling algorithms on long jobs. To this end, we selected top 20% longest jobs and analyzed their waiting times and warm-up times. Figure 8(b) shows that ONA penalizes long jobs less than SJF in terms of both waiting and warm-up times. ONA results indicate that about 35% of the long jobs having longer waiting times compared to the case of FIFO, but in case of SJF 45% of the jobs have such longer waiting times.

**Impact of inaccurate task processing time estimation.** We use estimated task processing times with errors to select tasks or jobs that should be scheduled on machines. Estimation error has no influence on FIFO scheme because FIFO schedules jobs by the order of job arrivals, not by task/job size. Figure 8(c) shows the influence of estimation errors on the performances of different schemes. The completion time on y-axis is a relative one to FIFO’s completion time, represented as percentage. We can see that both ONA and SJF are less susceptible to estimation error. Estimation of individual task processing time gets worse as estimation error increases, whereas the original order in execution sequence is relatively well preserved in case of ONA and SJF even under high estimation error condition. On the contrary, STF shows different pattern; as estimation error increases, the completion time of STF gradually increases, causing almost 7.5% increase in completion time between with an estimation error of 100%. STF is directly affected by erroneous task processing time estimation in a sense that it selects tasks which results in suboptimal completion time due to estimation error.

## IV. RELATED WORK

While assigning and scheduling jobs in distributed and parallel systems is a well-established area [8], [9], [10], scheduling in MapReduce framework is relatively less studied.

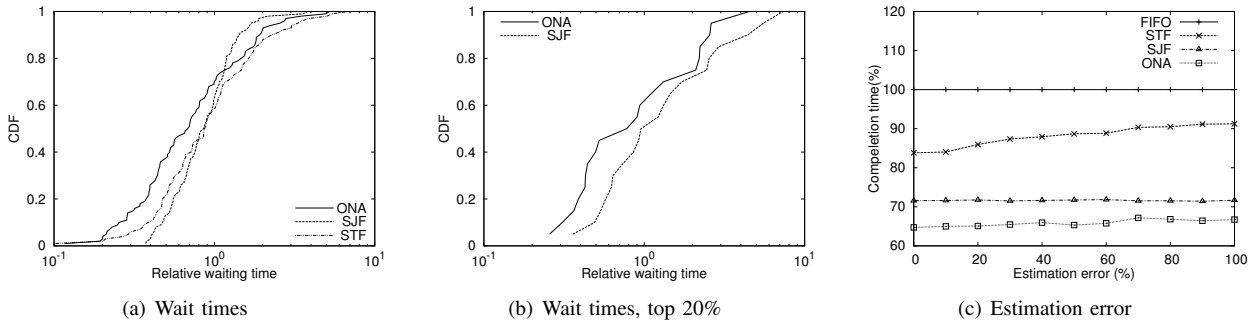


Fig. 8. Subplot(a) shows the CDF of relative waiting times of jobs compared to FIFO. Subplot (b) shows the CDF of wait times for the top 20% jobs. Subplot (c) shows the impact of processing time estimation error on the job completion time.

Quincy [2] and delay scheduling [3] are two recent schemes focused on fair scheduling in MapReduce-like systems. Both these scheduling algorithms model many aspects of real systems, for example, to preserve locality of computation and data as much as possible, and ensure fairness across jobs—which we do not consider in our model to focus on the pure scheduling problem. Their goal, however, is not so much finding an optimal solution, or even understanding how close to optimal their schemes are. In contrast, our work also focuses on understanding the optimality aspects of scheduling in this framework. LATE [11] is another practical algorithm that addresses the performance issues incurred by speculative execution of MapReduce in heterogeneous environments, where the assumption about linear progress of tasks is not satisfied.

A recent work by Sandholm *et al.* in [12] provides automatic application-independent optimization strategies by prioritizing users, stages in a job, and bottleneck components within a stage. While their system allows users to freely configure the priority of an application or different priorities on different components, users’ freedom is regulated by the limited total priority that users can assign.

The closest works to the scheduling problem considered in this paper are the single machine scheduling problems to minimize the sum of weighted completion times. The polyhedron of necessary conditions for the single machine problem was derived in [5]. This result was used by [6] to derive approximation algorithms for single machine weighted completion time with additional side constraints. Better approximation algorithms for the single machine problem were derived in [13]. A 2-competitive algorithm for the single machine online scheduling problem was derived in [14]. The multi-processor scheduling literature also considers several related problems but not the scheduling problem that we address in this paper.

## V. CONCLUSION

Large-scale data processing needs of many enterprises are routinely met using distributed processing in clusters. MapReduce has emerged as a popular programming model for distributing these processing jobs across many nodes in a cluster. Typical enterprises require running many MapReduce jobs simultaneously in the data center, often involving many more tasks than can be simultaneously satisfied. A scheduling

algorithm therefore is critical in determining which jobs to schedule at what times. In this paper, we focus on a theoretical model for this scheduling problem. In particular, we formulate a linear program that minimizes the job completion times to solve the problem. Given the hardness at solving the LP, we devise approximate algorithms that achieve feasible schedules within a small constant factor of the optimal value of the objective function. Using simulations, we show that our algorithm outperforms many traditional strategies such as FIFO, SJF and STF consistently across different workloads.

## REFERENCES

- [1] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *USENIX OSDI*, 2004.
- [2] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: Fair Scheduling for Distributed Computing Clusters,” in *SOSP*, 2009.
- [3] 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.
- [4] 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.
- [5] M. Queyranne, “Structure of a simple scheduling polyhedron,” *Mathematical Programming*, vol. 58, no. 2, pp. 263–285, 1993.
- [6] A. S. Schulz, “Scheduling to minimize total weighted completion time: Performance guarantees of lp-based heuristics and lower bounds,” 1995.
- [7] N. Garg and J. Knemann, “Faster and simpler algorithms for multicommodity flow and other fractional packing problems,” in *In Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, 1998, pp. 300–309.
- [8] B. W. Lampson, “A scheduling philosophy for multiprocessing systems,” *Communications of the ACM*, vol. 11, pp. 347–360, 1968.
- [9] M. Harchol-Balter, “Task assignment with unknown duration,” *Journal of the ACM*, vol. 49, no. 2, pp. 260–288, 2002.
- [10] M. E. Crovella, M. Harchol-Balter, and C. D. Murta, “Task assignment in a distributed system (extended abstract): improving performance by unbalancing load,” in *SIGMETRICS ’98*, 1998, pp. 268–269.
- [11] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments,” in *USENIX OSDI*, 2008.
- [12] T. Sandholm and K. Lai, “Mapreduce optimization using regulated dynamic prioritization,” in *SIGMETRICS ’09*, 2009.
- [13] M. X. Goemans, “Improved approximation algorithms for scheduling with release dates,” in *SODA ’97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, 1997, pp. 591–598.
- [14] E. J. Anderson and C. N. Potts, “On-line scheduling of a single machine to minimize total weighted completion time,” in *SODA ’02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2002, pp. 548–557.