Joint Optimization of Overlapping Phases in MapReduce

Minghong Lin
Computer Science
California Institute of Technology

Li Zhang
IBM T.J. Watson Research Center

Adam Wierman
Computer Science
California Institute of Technology

Jian Tan
IBM T.J. Watson Research Center

ABSTRACT

MapReduce is a scalable parallel computing framework for big data processing. It exhibits multiple processing phases, and thus an efficient job scheduling mechanism is crucial for ensuring efficient resource utilization. This paper studies the scheduling challenge that results from the overlapping of the "map" and "shuffle" phases in MapReduce. We propose a new, general model for this scheduling problem, and validate this model using cluster experiments. Further, we prove that scheduling to minimize average response time in this model is strongly NP-hard in the offline case and that no online algorithm can be constant-competitive. However, we provide two online algorithms that match the performance of the offline optimal when given a slightly faster service rate (i.e., in the resource augmentation framework). Finally, we validate the algorithms using a workload trace from a Google cluster and show that the algorithms are near optimal in practical settings.

1. INTRODUCTION

MapReduce is a scalable parallel computing framework for big data processing that is widely used in data centers for a variety of applications including web indexing and data analysis. MapReduce systems such as Hadoop [1], Dryad [20], and Google's MapReduce [8] have been deployed on clusters containing as many as tens of thousands of machines in a wide variety of industry settings.

The foundation of the MapReduce framework is that each job consists of multiple dependent phases, where each phase consists of multiple tasks that are run in parallel. In particular, each job must move through a map phase, a copy/shuffle phase, and a reduce phase. The map and reduce phases typically perform the bulk of the data processing, while the shuffle phase is typically a data transfer phase. Crucially, tasks in each phase depend on results computed by the previous phases. As a result, scheduling and resource allocation is a complex and crucial task in MapReduce systems.

Though MapReduce has been adopted widely, there are still significant opportunities for improvement in the scheduling and optimization of MapReduce clusters. As a result, the design of mechanisms for improving resource allocations, and thus performance, in MapReduce systems is an active research area in academia and industry, e.g., [3–5, 14, 17, 21, 25, 29, 31, 39]. Examples of such research include designing schedulers to efficiently handle "outliers" and "stragglers" [4], to provide improved locality during the map phase [14, 17, 25], and to reduce "starvation" [29, 39].

Most of the focus in research on MapReduce optimizations is on the data processing phases (map and reduce); however the completion time of a job in MapReduce is also highly dependent on the data transfer phase (shuffle) since it takes non-negligible time to transfer intermediate data. In fact, for common MapReduce jobs, most of the time is spent in the map phase and shuffle phases, e.g., according to [22, 40], only around 7% of the workload in a production MapReduce cluster are reduce heavy jobs.

To reduce the time for data transfer, a common technique is to move the computation close to the data. For example, Delay Scheduling [39] in Hadoop prefers to allocate map slots to map tasks with data on the same server/rack. By using this technique, more than 90% of data fetching for the map phase can be eliminated. However, the shuffle phase, which performs the data fetching for the reduce phase, hardly benefits from this technique since each reduce task has to fetch data from all map tasks. Thus, in order to improve performance, the shuffle phase needs to be coordinated with the map phase. Such coordination is typically non-existent in current MapReduce schedulers, which usually make job scheduling decisions in each phase individually, e.g., FIFO scheduler, Fair Scheduler [1], and Capacity Scheduler [2] in Hadoop.

The importance of coordination between the map and shuffle phases is further highlighted by the fact that the map phase and the shuffle phase can overlap each other. The Hi-Tune [7] analyzer highlights how critical this overlapping is for overall MapReduce performance: this overlapping helps to ensure that both the CPU and the network are highly utilized, preventing either from being the bottleneck whenever possible. By default, Hadoop allows the shuffle phase to start when the progress of the map phase exceeds 5%. This means that for a given job, some map tasks will have finished and had the corresponding intermediate data transferred to its reduce tasks, while other map tasks are still in the processing.

Summary of contributions.

In this work, we focus on a challenging and important scheduling problem in the MapReduce framework that has received little attention so far: *the scheduling of the map and shuffle phases given phase overlapping.*

To this end, **our first contribution is the development of a model for studying the phase overlapping of the map and shuffle phases**. Note that the particular form of the phase overlapping in MapReduce makes data processing different from the traditional multiple stage processing models in manufacturing and communication networks [16,24,38]. In particular, in a typical tandem queueing model [38] (or flowshop model [16,24]) a job must complete processing at one station before moving to the following station. In contrast, in MapReduce a job may have tasks that start the shuffle phase before all the tasks complete the map phase. We present the details of our overlapping tandem queue model in Section 2, where we also use implementation experiments on a MapReduce cluster to validate the model (Figure 4).

Note that at the level of tasks, a MapReduce system may be modeled as a fork-join queueing network, e.g., [28, 33, 41]; however, such approaches do not necessarily faithfully capture the particular features of MapReduce and they yield complicated models in which it is difficult to design and analyze job level scheduling policies. Our approach is to model the job-level processing in order to provide a simpler (still accurate) model, that clearly exposes the scheduling challenge created by phase overlapping.

Given our new model for phase overlapping in MapReduce, **our second contribution is the design and analysis of new scheduling policies that efficiently make use of phase overlapping in order to minimize the average response time.** The task of scheduling in the presence of phase overlapping is difficult. If one seeks to minimize the average response time, it is desired to give priority to jobs with small sizes (whenever possible) within each of the map and shuffle queues, since Shortest Remaining Processing Time (SRPT) first scheduling minimizes average response time within a single server [27]. However, because of phase overlapping, it is also necessary to ensure that map tasks with large shuffle sizes are completed first in order to avoid wasting capacity at the shuffle queue. These objectives are often competing, and thus it is difficult to balance them without knowing information about future arrivals.

Our first set of analytic results (Section 3) highlight the challenging nature of this scheduling problem. In particular, we prove that the task of minimizing response time in this model is strongly NP-hard, even in the case of batch arrivals (all jobs arrive at time zero) when all job sizes are known and all jobs having identical map sizes. Further, ignoring computational constraints, we prove that when jobs arrive online, no scheduling policy can be "constant-competitive". In particular, in the worst case, the ratio of the average response time of any online algorithm and the average response time of the offline optimal solution is lower bounded by $\Omega(n^{1/3})$, where n is the number of jobs in the instance. Thus, the ratio can be unbounded. Importantly, this result highlights that it is "exponentially harder" to schedule overlapping phases than it is to schedule a multi-server queue, where it is possible to be $O(\min\{\log P, \log n/m\})$-competitive using SRPT scheduling [23], where m is the number of servers and P is the ratio of maximum processing time to minimum processing time.

Despite the difficulty of scheduling in the presence of phase overlapping, *our second set of results (Section 4) provide algorithms with strong worst-case performance guarantees.* In particular, we give two algorithms, MaxSRPT and SplitSRPT, and prove worst-case bounds on their performance in both the batch setting and the online setting. In the batch setting we prove that both algorithms are at worst 2-competitive, and that they are complementary in the sense that MaxSRPT is nearly optimal when the ratio of map sizes to shuffle sizes of jobs tends to be close to 1 and SplitSRPT is nearly optimal when the ratio of map sizes to reduce sizes of jobs tends to be far from 1. In the online setting, we cannot expect to attain constant bounds on the competitive ratio, so we focus instead on providing "resource augmentation" analysis. In particular, we prove that our algorithms can match the offline optimal average response time when given slightly more capacity. Specifically, both algorithms are 2-speed 1-competitive in the worst case, and again MaxSRPT is near-optimal when map and shuffle sizes are "balanced" and SplitSRPT is near optimal when map and shuffle sizes are "unbalanced".

In addition to providing worst-case analysis of MaxSRPT and SplitSRPT, we study their performance in practical settings using trace-based simulations in Section 5. Specifically, using Google MapReduce traces [37] we show that MaxSRPT and SplitSRPT provide a significant improvement over Limited Processor Sharing (LPS) scheduling, which is a model of Fair Scheduler in Hadoop. Further, we show that on these realistic traces MaxSRPT and SplitSRPT have mean response time close to the optimal offline scheduler. Finally, since unfairness is always a worry when using policies based on SRPT scheduling, we provide a study of unfairness under MaxSRPT and SplitSRPT. Similarly to the analysis of SRPT in a single server queue [11, 12, 26, 34–36], our results highlight that unfairness is not a major worry under these policies.

2. MODELING PHASE OVERLAPPING

Our focus in this paper is on understanding how to efficiently schedule map and shuffle tasks given the presence of phase overlapping. To approach this problem formally, we first need to define and validate a model, which will then serve as the basis for the design and analysis of scheduling algorithms. In this section we first describe our analytic model, that we term an "overlapping tandem queue" and then validate the model using experiments on a cluster testbed.

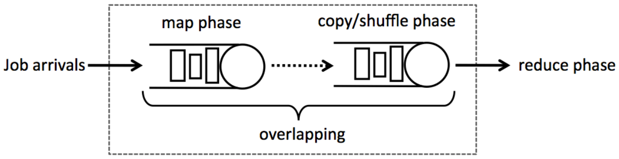
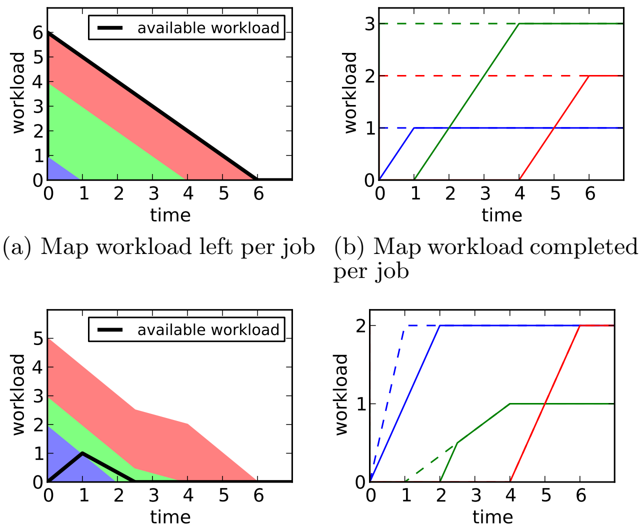


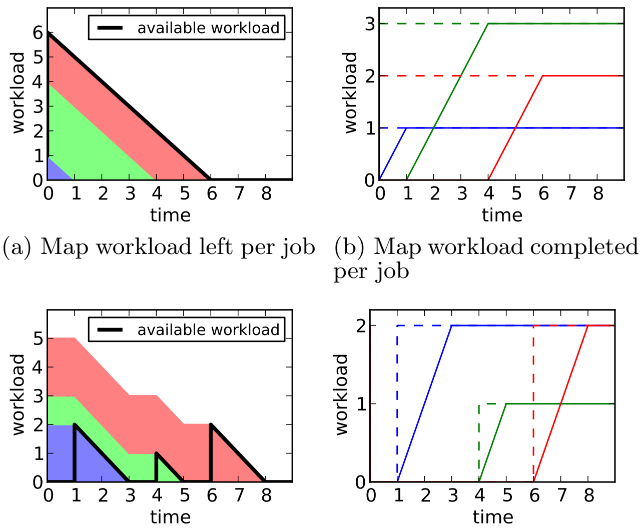
Figure 1: Overview of the overlapping tandem queue model for phase overlapping in MapReduce.



(a) Map workload left per job (b) Map workload completed per job

(c) Shuffle workload left per job (d) Shuffle workload completed per job

Figure 2: Workload progression in a overlapping tandem queue model after the arrival of three jobs: $J1$ (blue), $J2$ (green), $J3$ (red).



(a) Map workload left per job (b) Map workload completed per job

(c) Shuffle workload left per job (d) Shuffle workload completed per job

Figure 3: Workload progression in a traditional tandem queue model after the arrival of three jobs: $J1$ (blue), $J2$ (green), $J3$ (red).

2.1 The overlapping tandem queue model

The model we propose is at the "job level" and our goal is to develop scheduling algorithms that minimize the average response time, i.e., the time between the arrival to the map phase and the completion at the shuffle phase.

Let us start with the workload. In a typical MapReduce cluster, each job is split into hundreds, or even thousands, of tasks. To develop a job level model, we ignore the number of tasks into which a job is divided and simply consider a job as certain amount of map workload (the map job size) and shuffle workload (the shuffle job size). Thus, an instance consists of n jobs $J_1, \dots, J_n$, where each has a release/arrival time r_i and a pair of job sizes (x_i, y_i) where x_i is the map job size (i.e., computation needed) and y_i is the shuffle job size (i.e., intermediate data size needing to be transferred).

The MapReduce system itself is modeled by a map queue and a shuffle queue as shown in Figure 1. Denote the map service capacity of the cluster by μ_m (i.e., total processing speed of all map slots) and the shuffle service capacity μ_c the shuffle capacity of the cluster (i.e., total network capacity for transferring intermediate data from map slots to reduce slots). Without loss of generality, we let $\mu_m = 1$ and $\mu_c = 1$ and normalize the job sizes accordingly.

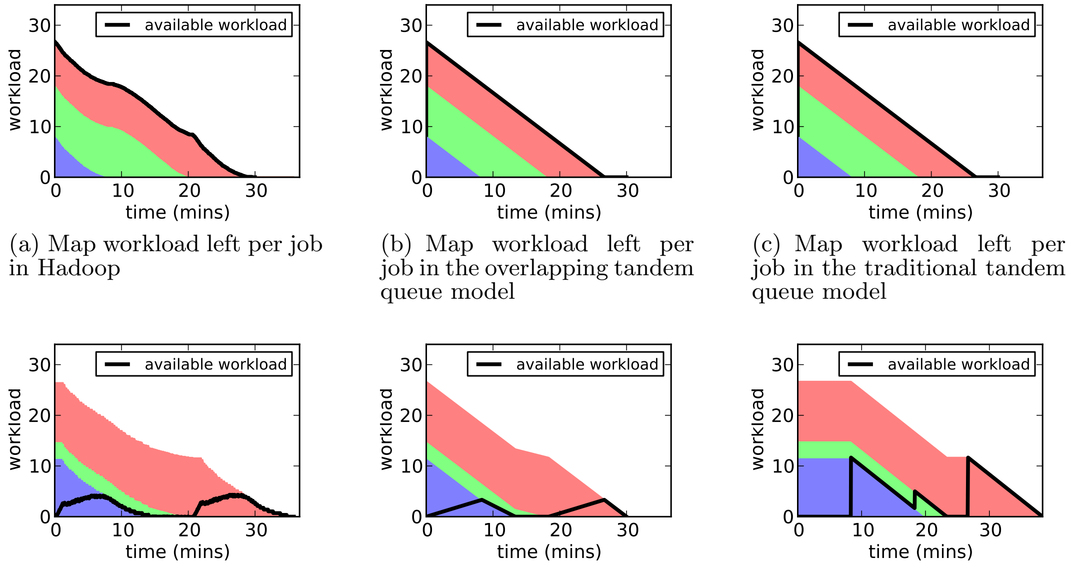
Each job has to go through the map queue and shuffle queue; however the two phases may overlap each other, i.e., part of the job may enter shuffle queue before the entire job is finished at the map queue. This overlapping is not arbitrary, we require that *the fraction of the workload of each job that has arrived to the shuffle queue at time t is not more than the fraction of the workload of the same job completed at the map phase at time t*. This constraint comes from the fact that pieces of intermediate data are available to the shuffle phase only after they has been generated from the map phase. Once the entire job has finished at the shuffle phase, the job leaves the system we considered.

Phase overlapping is the key difference of our model compared to the traditional tandem queue models. This overlapping allows us to model the fact that, as tasks finish, the shuffle phase for that task can proceed even though other tasks of the same job may not have completed the map phase. Note that, by default, Hadoop allows the shuffle phase to start when the progress of the map phase exceeds 5% [15]. This phase overlapping can be thought of a software-level parallel pipeline operating at the task level, and our model provides a job level (coarse) view of this process. Importantly, this context is very different from instruction level pipelining used in computer architecture, in which the pipeline is tightly coupled and controlled at an extremely fine degree of granularity.

To define the overlapping tandem queue model more formally, denote the job size of J_i at time $t \in [r_i, \infty)$ by $(x_i(t), y_i(t))$, i.e., the remainning job size, and define $1_{y_i(t)>0}$ to be an indicator function representing whether $y_i(t)$ is nonzero. Using this notation, we can state the scheduling objective and model constraints as follows:

$$\begin{aligned}
\text{minimize} \quad & \int_0^\infty \sum_{r_i \le t} 1_{y_i(t)>0} dt \\
\text{subject to} \quad & \sum_{r_i \le t} dx_i(t)/dt \ge -1, \sum_{r_i \le t} dy_i(t)/dt \ge -1 \\
& dx_i(t)/dt \le 0, dy_i(t)/dt \le 0 \qquad (1) \\
& x_i(r_i) = x_i, y_i(r_i) = y_i \\
& y_i(t)/y_i \ge x_i(t)/x_i \ge 0,
\end{aligned}$$

where the objective is the sum of response times, the first two constraints bound the service rate of the map and shuffle queues, the third constraint provides the release/arrival times of the jobs, and the final constraint captures the fact that the progress in the shuffle phase is not faster than the progress in the map phase for each job.



(a) Map workload left per job in Hadoop

(b) Map workload left per job in the overlapping tandem queue model

(c) Map workload left per job in the traditional tandem queue model

(d) Shuffle workload left per job in Hadoop

(e) Shuffle workload left per job in the overlapping tandem queue model

(f) Shuffle workload left per job in the traditional tandem queue model

Figure 4: Workload progression in a Hadoop MapReduce cluster, the overlapping tandem queue model and the traditional tandem queue model after the arrival of three jobs: $J1'$ (blue), $J2'$ (green), $J3'$ (red).

An example.

To illustrate how the overlapping tandem queue works, consider a simple case with three jobs in the system. $J1$(blue), $J2$(green), $J3$(red) have job sizes $(1,2)$, $(3,1)$ and $(2,2)$ respectively. To get a clear illustration, assume both phases work on jobs in the order of $J1$, $J2$ and $J3$. We show the workload progression of the two stations in Figure 2. The map workload progression (Figure 2(a)) is the same as in a single server queue model with First In First Out scheduling, where the total (available) workload (shown by the thick black curve) is drained at maximum rate 1 because of the work conserving policy. The shuffle workload progression (Figure 2(c)) is more complex. During $t \in [0,1]$, since J_1 is shuffle-heavy ($y_1 > x_1$), the available shuffle workload accumulates (shown by the thick black curve) and the total shuffle workload is drained at maximum rate 1. At $t = 1$, J_1 is done at the map phase and has 1 unit of workload left at the shuffle phase. After $t = 1$, since J_2 is map-heavy, the available shuffle workload buffered is decreasing and finally drained out at $t = 2.5$, when J_2 has 1.5 units of workload done at the map phase and 0.5 units of workload done at the shuffle phase. During this period ($t \in [1, 2.5]$), the total shuffle workload to be completed is still drained at maximum rate 1. After $t = 2.5$, the total shuffle workload can only be drained at rate 1/3, which is the rate the available shuffle workload of J_2 generated from the map phase, and therefore no workload is accumulated at the shuffle phase (as shown by the thick black curve). This draining rate lasts until $t = 4$ when J_2 is done at both the map phase and the shuffle phase. During $t \in [4, 6]$, the shuffle phase is draining the shuffle workload at maximum rate 1 and there is no shuffle workload accumulated since $y_3 = x_3$ for J_3. The supplemental figures Figure 2(b) and 2(d) show the map progress and the shuffle progress for each job separately, where the actual progress is represented by a solid curve (blue for J_1, green for J_2 and red for J_3), and the workload ready for processing is represented by a dashed curve.

Contrast with tandem queues.

The overlapping tandem queue model that we introduce here is clearly related to the traditional tandem queue model that has been studied in depth in the queueing and the scheduling communities [13, 30]. However, the phase overlapping that we model means that the overlapping tandem queue is fundamentally different from a traditional tandem queue model. To highlight the differences, we show the workload progression of the two queues of a tandem queue in Figure 3. The contrast between Figure 3 and Figure 2, which shows the workload progression in the overlapping tandem queue, is stark. In a traditional tandem queue a job cannot be processed at the shuffle queue until it completes at the map queue; while in the overlapping tandem queue the job can simultaneously be processed in both queues. Further, when one considers the design of scheduling policies for a overlapping tandem queue, the form of phase overlapping in the model implies that scheduling in the first queue has an unavoidable impact the performance of the second queue.

2.2 Model validation

In the following, we use experiments in a MapReduce cluster in order to provide some validation of the model of phase overlapping in MapReduce that we present above. The cluster we use for experimentation has 15 servers, and each server is configured with 4 map slots and 2 reduce slots. The cluster runs Hadoop v.0.22.

Using this cluster, we have run a variety of experiments in order to explore the alignment of the model presented above with real experimental sample paths. For presentation here we include only one representative example. In particular, Figure 4 shows the workload progression in our

MapReduce cluster after the arrival of three jobs and contrasts this with the evolution in our model. In this example, the Hadoop cluster uses the default FIFO scheduler. For the corresponding overlapping tandem queue model, we also use FIFO scheduling at the map station, and use Process Sharing (PS) scheduling to mimic the network sharing at the shuffle station. The important point to note is the alignment of the workload sample path behavior in Hadoop (Figure 4(d)) and the overlapping tandem queue model (Figure 4(e)), including the kink in the service rate of the shuffle queue. In contrast, as shown in Figure 4(f), the evolution in the traditional tandem queue model is quite different from that in the Hadoop experiment.

More specifically, the experiment summarized in Figure 4 was designed as follows. Three jobs are submitted to the cluster in a batch. J_1' (blue) is the Hadoop default sort example with 22 GB of input (random records). It is split into 352 map tasks and configured with 15 reduce tasks (one for each server). Running alone in the idle cluster without phase overlapping, it spends about 7 minutes in the map phase and 11 minutes in the shuffle phase; thus we use job size $(7, 11)$ for J_1' in our model. J_2' (green) is the Hadoop default grep example with 28 GB of input (Wikipedia Free Encyclopedia). It is split into 427 map tasks and configured with 15 reduce tasks (one for each server). Running alone in the idle cluster without phase overlapping, it spends about 10 minutes in the map phase and 3 minutes in the shuffle phase, thus we use job size $(10, 3)$ for J_2' in our model. J_3' (red) is again the Hadoop default sort example with the same configuration and similar input as J_1', thus we use job size $(10, 3)$ for J_3' in our model.

3. THE HARDNESS OF SCHEDULING A OVERLAPPING TANDEM QUEUE

To begin our study of scheduling in the overlapping tandem queue model we focus on understanding lower bounds on what is achievable, i.e., hardness results for the scheduling problem.

Our analysis reveals that the problem is hard in two senses. First, computing the optimal schedule in the *offline setting*, i.e., when all information about arrival times and job sizes are known ahead of time, is strongly NP-hard[1]. This hardness result holds even if all jobs arrive at time zero (the batch setting) and all map sizes are identical. Second, in the *online setting*, i.e., job size and arrival time information is only learned upon arrival of the job, no scheduling algorithm can achieve an average response time within a constant factor of the optimal offline schedule.

3.1 Offline scheduling

In the offline scheduling formulation, it is assumed that all information about job arrival times (release times) and job sizes are known to the algorithm in advance. Note that the offline scheduling problem is not just of theoretical interest, it is also practically important because of the batch scheduling scenario in which jobs are submitted to a MapReduce cluster in a "batch" and the algorithm does offline scheduling in order to clear all jobs.

Given an offline scheduling problem, the natural question is how to compute the optimal scheduling policy efficiently. Unfortunately, in the overlapping tandem queue it is strongly NP-hard to minimize the average response time. Though, on the positive side, in Section 4 we give two simple algorithms that are always within a factor of two of optimal.

Theorem 1. *In the overlapping tandem queue model it is strongly NP-hard to decide whether there exists a schedule with average response time no more than a given threshold, even if the arrival times and map sizes of all jobs are identical.*

To provide context for this result, it is useful to consider the related literature from the flowshop scheduling community, e.g., [10, 16, 19, 24]. In particular, the flowshop literature considers traditional tandem queues (which do not allow overlapping) with all jobs arrive at time zero and tend to focus on a different performance metric: makespan. The makespan is the time between the arrival of the first job and the completion of the last job. For makespan, results are positive in the overlapping tandem queue model: the makespan can be minimized by scheduling jobs with $y_i \geq x_i$ before jobs with $y_i < x_i$ at both stations.

The detailed proof of Theorem 1 is in the appendix. It is based on a polynomial-time reduction from the *Numerical Matching with Target Sums (NMTS)* problem [9], which is known to be strongly NP-hard. The NMTS problem is defined as follows.

Definition 1. ***Numerical Matching with Target Sums (NMTS):*** *Consider three sets $F = \{f_1, \ldots, f_q\}$, $G = \{g_1, \ldots, g_q\}$ and $H = \{h_1, \ldots, h_q\}$ of positive integers, where $\sum_{i=1}^{q}(f_i + g_i) = \sum_{i=1}^{q} h_i$. Does there exist permutations π and σ such that $f_{\pi(i)} + g_{\sigma(i)} = h_i$ for $i = 1, \ldots, q$?*

It is interesting that our numerical optimization problem in a continuous domain can be reduced from a combinatorial optimization problem in a discrete domain. The basic idea is that, for some input in the overlapping tandem queue model, the optimal job completion order satisfies a certain property. Further, finding a completion order satisfying this property is strongly NP-hard, no matter how the capacities are shared during the job processing.

3.2 Online scheduling

In the online setting, the release time of a job is the first time the scheduler is aware of the job information, including its release time and size information. Thus, the online scheduling problem is usually harder than the offline scheduling problem. In our case, we have seen that the offline scheduling problem is hard because of computational constraints, here we show that the online scheduling problem is hard because of the lack of knowledge about future arrivals.

A widely used metric to quantify the performance of an online algorithm is the *competitive ratio*, which is the worst case ratio between the cost of the online algorithm and the cost of the offline optimal solution. The theorem below highlights that no online scheduling algorithm can have a constant competitive ratio in the overlapping tandem queue model.

[1] A problem is said to be strongly NP-hard (NP-hard in the strong sense), if it remains so even when all of its numerical parameters are bounded by a polynomial in the length of the input. For example, bin packing is strongly NP-hard while the 0-1 Knapsack problem is only weakly NP-hard.

Theorem 2. *In the overlapping tandem queue model, any online scheduling algorithm (deterministic or randomized) is at least $\Omega(n^{1/3})$-competitive, where n is the number of jobs for the instance.*

The theorem above highlights that, not only is the competitive ratio for any online algorithm unbounded, but the competitive ratio is lower bounded by a polynomial function of the input. As a comparison, SRPT scheduling is optimal in a single server queue, i.e., SRPT is 1-competitive. In contrast, in a multi-server queue with m identical servers the competitive ratio of SRPT is unbounded, but is upper bounded by $O(\min\{\log P, \log n/m\})$ [23], where P is the ratio of maximum processing time to minimum processing time. The comparison with Theorem 2 highlights that scheduling in the overlapping tandem queue is "exponentially harder" than the online scheduling problem with multiple identical machines.

The proof of Theorem 2 is given in the appendix, and highlights the difficulty of the online scheduling in the overlapping tandem queue model. In particular, at both queues, scheduling in a manner close to SRPT is desirable in order to avoid forcing small jobs to queue behind larger jobs (a crucial part of minimizing average response time). However, it is also important to ensure that capacity is not wasted at the shuffle (second) queue. These goals can be in conflict, and resolving the conflict optimally depends on information about future arrivals.

4. ALGORITHM DESIGN

Given the results in the previous section highlighting the difficulty of scheduling in the overlapping tandem queue model, we cannot hope for efficient, optimal algorithms. However, we can hope to be near-optimal. In this section, we provide two algorithms (MaxSRPT and SplitSRPT) that are near optimal in two senses: In the *batch* setting they guarantee average response time within a factor of two of optimal, i.e., a constant approximation ratio; and in the *online* setting they guarantee to match optimal average response time when given twice the service capacity, i.e., they are 2-speed 1-competitive. These worst-case guarantees are a key contrast to the heuristic approaches typically suggested. Further, we actually prove stronger guarantees on the two algorithms that highlight that MaxSRPT is nearly optimal if the map and shuffle sizes of jobs are "balanced" and SplitSRPT is near optimal if the map and shuffle sizes of jobs are "unbalanced".

In the following we define and analyze the two algorithms, and then end the section with a brief discussion of considerations regarding the practicality of these algorithms for implementation in MapReduce clusters.

4.1 MaxSRPT

It is well known that SRPT minimizes the average response time in a single server queue [27], and so it is a natural algorithm to try to extend to our model. For a single server queue, SRPT always schedules the job whose remaining processing time is the smallest, breaking ties arbitrarily. The intuition for its optimality is that SRPT pumps out jobs as fast as possible and thus minimizes the average number of jobs in the system, which is linearly related to the average response time. Similarly, we would like to pump out jobs as fast as possible in the overlapping tandem queue model; however it is not clear what the "remaining size" should be in this setting.

Our approach is to think of the "remaining processing time" of a job as the maximum of its remaining map size and remaining shuffle size. Intuitively, this is natural since this quantity is the time it takes for the job to finish in an idle system. This notion of "remaining processing time" motivates our definition of MaxSRPT:

Algorithm 1 (MaxSRPT). *At time t, each station processes a job J_i having the minimum $\max(x_i(t), y_i(t))$, where $x_i(t)$ and $y_i(t)$ are the remaining map size and the remaining shuffle size of job J_i at time t.*

In MaxSRPT, the first station (modeling the map phase) is work conserving; however, the second station (modeling the shuffle phase) is not necessarily work conserving. For the performance guarantee we prove below, we just require the second station to be work conserving for the available shuffle workload from the job at the head of the queue. Of course, in a real system (and in the experimental results in Section 5), it is beneficial to keep the second station work conserving by working on any available shuffle workload with unused capacity.

Performance analysis.

We now move to the performance analysis of MaxSRPT. We provide two *worst-case* results characterizing the performance. First, we start with a bound on the performance in the online setting. In this setting, we have already seen that no online algorithm can be constant competitive, and so we focus on "resource augmentation" analysis, i.e., understanding how much extra capacity is necessary for MaxSRPT to match the performance of the offline optimal. The following theorem shows that MaxSRPT can match the optimal average response time using only a small amount of extra capacity.

Theorem 3. *Denote $\alpha = \max_i \max(x_i/y_i, y_i/x_i)$. MaxSRPT is $2\alpha/(1+\alpha)$-speed 1-competitive.*

By saying $2\alpha/(1+\alpha)$-speed 1-competitive, we mean that the algorithm running with service capacity $2\alpha/(1+\alpha)$ at both stations achieves average response time not more than that of the offline optimal algorithm running with service capacity 1 at both stations. This is called a "resource augmentation" guarantee, and is a widely used mechanism for studying difficult offline or online problems.

Note that Theorem 3 guarantees that MaxSRPT is at worst 2-speed 1-competitive. However it also highlights that the "balance" of the job sizes affects the performance of MaxSRPT. In particular, α is small if the map sizes and the shuffle sizes are similar for each job (though the sizes can be very different for different jobs). If jobs tend to be balanced, then MaxSRPT is nearly optimal.

Proof of Theorem 3. Denote J_i the i^{th} job arrival with size (x_i, y_i) released at time r_i in the overlapping tandem queue model. Consider the overlapping tandem queue model with processing speed 1 at both stations and compare it to a single server queue model with processing speed 2. For job arrival J_i in the overlapping tandem queue model, construct a job arrival with size $x_i + y_i$ releasing at r_i in the single server queue model. Assume J_i under the optimal scheduling has processing rate at the map phase $p_i(t)$ and processing rate at the shuffle phase $q_i(t)$ at time t. Then in the single server queue model, the corresponding job is processed at rate $p_i(t) + q_i(t)$ at time t. Obviously the total response time in the two systems are exactly the same.

Now, we scale both the server speed and job sizes for the single server queue model by a factor of $\alpha/(1+\alpha)$, which does not change the total response time. As a result, we have a single server queue with processing speed $2\alpha/(1+\alpha)$ and job sizes $(x_i + y_i)\alpha/(1+\alpha)$. Notice that $\max(x_i, y_i) \leq (x_i + y_i)\alpha/(1+\alpha)$, and SRPT is optimal for minimizing mean response time in the single server queue model. Denote J_i' the i^{th} job arrival to a single server queue with size $z_i = \max(x_i, y_i)$ released at time r_i. Then the total response time using SRPT serving $\{J_i'\}$ in the single server queue with speed $2\alpha/(1+\alpha)$ is not more than the total response time using the optimal scheduling policy serving $\{J_i\}$ in the overlapping tandem queue model with speed 1 at both stations.

Next, consider the overlapping tandem queue model serving $\{J_i\}$ with speed $2\alpha/(1+\alpha)$ at both stations and the single server queue model serving $\{J_i'\}$ with speed $2\alpha/(1+\alpha)$. Denote $L_i(t) = \max(x_i(t), y_i(t))$ in the overlapping tandem queue model. Then the arrival of J_i in the overlapping tandem queue model and the arrival of J_i' in the single server queue model bring equal amount of L_i and z_i. For MaxSRPT, the minimum $L(t)$ is always decreasing at rate $2\alpha/(1+\alpha)$ at any time t and some other $L(t)$ may also decrease simultaneously. In the single server queue model, the minimum $z(t)$ is decreasing at rate $2\alpha/(1+\alpha)$ and no other $z(t)$ decreases. Therefore, MaxSRPT in the overlapping tandem queue model finishes no fewer jobs than SRPT in the single server queue model at all times. It follows that the total response time of MaxSRPT is not bigger.

In summary, MaxSRPT with speed $2\alpha/(1+\alpha)$ has total response time not more than the optimal scheduling policy with speed 1. □

An important special case to study is the case of batch arrivals, i.e., when all jobs arrive at the same time. Recall that minimizing the average response time is strongly NP-hard even in this setting.

In the batch setting, MaxSRPT simply sorts the jobs in ascending order based on $L_i = \max\{x_i, y_i\}$. And then both stations work on jobs in this order, subject to data availability on the second station. Since all jobs arrive at the same time and there are no future arrivals, scaling the speed is equivalent to scaling the time. Therefore, we obtain an approximation ratio (ratio of average response time of MaxSRPT and that of the optimal schedule) for the batch setting immediately from the proof of Theorem 3. In particular:

Corollary 4. *Consider a batch instance of n jobs with job sizes (x_i, y_i) and release times $r_i = 0$ for $i = 1, \ldots, n$. Denote $\alpha = \max_i \max(x_i/y_i, y_i/x_i)$. MaxSRPT has an approximation ratio of $2\alpha/(\alpha+1)$, which is not more than 2.*

Note that the approximation ratio in Corollary 4 is tight for any given $\alpha > 1$ (and thus the resource augmentation result in Theorem 3 is also tight for any given $\alpha > 1$). To see this, consider an instance with $n/2$ of jobs with size $(\alpha - \epsilon, 1)$ where $\epsilon < \alpha - 1$ is a small positive constant, and $n/2$ of jobs with size $(1, \alpha)$. MaxSRPT schedules all jobs with size $(\alpha - \epsilon, 1)$ followed by all jobs with size $(1, \alpha)$. For big n, the total response time is $T \sim n^2\alpha/2 - 3n^2\epsilon/8$. For an alternative policy which schedules each job with size $(1, \alpha)$ followed by a job with size $(\alpha - \epsilon, 1)$, the total response time is $T' \sim n^2(1+\alpha)/4$. Thus the approximation ratio is not less than $T/T' \sim (2\alpha - 3\epsilon/2)/(1+\alpha)$ which can be arbitrarily close to $2\alpha/(1+\alpha)$ by picking ϵ small enough.

4.2 SplitSRPT

The reason that MaxSRPT has worse performance when jobs are more unbalanced (i.e., α is big), is that it does not distinguish between map-heavy jobs and shuffle-heavy jobs. Motivated by this observation, in this section we present an alternative algorithm that has the same worst-case performance bounds, but performs better when map and shuffle sizes are "unbalanced."

The basic idea behind this algorithm is to mix map-heavy jobs and shuffle-heavy jobs in order to ensure that the capacities at both stations are efficiently utilized. In particular, the algorithm we suggest explicitly splits the capacity into a piece for each of these types of jobs.

Algorithm 2 (SplitSRPT). *Denote $Q(t)$ the set of jobs in the system at time t. Upon job arrival or departure, update $\beta(t) = \min_{J_i \in Q(t)} \max(x_i/y_i, y_i/x_i)$ (x_i and y_i are the original map size and shuffle size of J_i), $\mu_1 = 1/(1+\beta(t))$ and $\mu_2 = \beta(t)/(1+\beta(t))$. Update $S_1 = \{J_k | J_k \in Q(t), x_k \geq y_k\}$ and $S_2 = \{J_k | J_k \in Q(t), x_k < y_k\}$. Process jobs in S_1 according to SRPT based on their remaining map size with capacity μ_2 in the map station and capacity μ_1 in the shuffle station. Process jobs in S_2 according to SRPT based on their remaining shuffle size with capacity μ_1 in the map station and capacity μ_2 in the shuffle station.*

Note that the parameter $\beta(t)$ in SplitSRPT measures the level of "imbalance" of the job sizes. Thus, $\beta(t)$ is big if the map size and the shuffle size are very different for each job, and $\beta(t)$ is small otherwise.

Performance analysis.

As with MaxSRPT, we provide two performance bounds for SplitSRPT: a resource augmentation bound for the online setting and an approximation ratio for the batch setting. Intuitively, SplitSRPT has better performance when jobs are more "unbalanced" (i.e., $\beta(t)$ is big); and the results below confirm this intuition.

Specifically, in the online setting, we have the following resource augmentation guarantee.

Theorem 5. *Denote $\beta = \min_t \beta(t)$. SplitSRPT is $(1 + 1/\beta)$-speed 1-competitive.*

Notice that we always have $\beta \geq 1$ and thus $1 + 1/\beta \leq 2$. So, the worst-case bound matches that of MaxSRPT: SplitSRPT with double service capacity has average response time not more than that under the offline optimal scheduling. But, in contrast with MaxSRPT, the extra speed required by SplitSRPT approaches zero as jobs become more "unbalanced". Thus the algorithms are complementary to each other.

PROOF OF THEOREM 5. Denote J_k the k^{th} job arrival with size (x_k, y_k) released at time t_k in the overlapping tandem queue model. Let us first focus on jobs in S_1. At any time t, $\beta(t) \leq x_k/y_k$ for any job from S_1, and the capacities to run the jobs in S_1 are μ_2 in the map station and μ_1 in the shuffle station which satisfies $\mu_2/\mu_1 = \beta(t)$.

The capacity at the shuffle station is alway fast enough to clear all the available shuffle workload not matter which job is being processed, i.e., the bottleneck is the map phase. Therefore, the response time for jobs in S_1 is equivalent to a single server queue using SRPT with (time-varying) server speed μ_2, where a job arrives with size x_k at r_k if $x_k \geq y_k$. Notice that the server speed is always not less than $\beta/(1+\beta)$ at any time t, thus the mean response time for jobs in S_1 is always not more than that in a single server queue using SRPT with speed $\beta/(1+\beta)$ with x_k released at r_k if $x_k \geq y_k$.

Now, consider the average response time of the jobs in S_1 under the optimal scheduling policy. It is not less than the average response time in a single server queue model using SRPT with speed 1, where a job arrives with size x_k at r_k. Therefore, by using speed $1 + 1/\beta$ in SplitSRPT, jobs in S_1 have mean response time not more than that under the optimal scheduling using speed 1.

For jobs in S_2, the same argument applies, and so the proof is complete. □

Once again, a bound on the approximation ratio of SplitSRPT follows immediately from Theorem 5. In parallel to Corollary 4, we have

Corollary 6. *Consider a batch instance of n jobs with job sizes (x_i, y_i) and release times $r_i = 0$ for $i = 1, \ldots, n$. Denote $\beta = \min_i \max(x_i/y_i, y_i/x_i)$. SplitSRPT has an approximation ratio of $1 + 1/\beta$, which is not more than 2.*

Notice that the approximation ratio in Corollary 6 is tight for any given $\beta > 1$ (and thus the resource augmentation result in Theorem 5 is also tight for any given $\beta > 1$). To see this, consider an instance with $n-1$ of jobs with size $(\beta, 1)$, and a job with size $(n-1, (n-1)\beta)$. SplitSRPT schedules the job $(n-1, (n-1)\beta)$ in parallel to other jobs. For big n, the total response time is $T \sim n^2(1+\beta)/2$. For an alternative policy which schedules all jobs with size $(\beta, 1)$ one by one, and then followed by the job with size $(n-1, (n-1)\beta)$. The total response time is $T' \sim n^2\beta/2$. Thus the approximation ratio is not less than $T/T' \sim 1 + 1/\beta$.

4.3 Implementation considerations

To end our discussion of algorithm design, it is important to highlight some of the practical issues involved in implementing MaxSRPT and SplitSRPT in a MapReduce cluster. Both of these algorithms are simple, and so are reasonable candidates for implementation; however there are two crucial aspects that must be addressed in order to make implementation possible: job size estimation and capacity sharing. We discuss each of these below.

Job size estimation: Both MaxSRPT and SplitSRPT rely heavily on having remaining size information for jobs. This issue has been discussed and addressed previously in the literature [6, 18], and some well-known methods have been adopted in Hadoop [32,39,40]. Briefly, such approaches work as follows. The job size for a map phase can be estimated by looking at historical data of similar jobs or by running a small fraction of the job (i.e., a few tasks) and using linear prediction [40]. Similarly, the job size for the shuffle phase can be estimated by looking at historical data of similar jobs or by running a few map tasks of the job, looking at the intermediate data size generated, and using linear prediction.

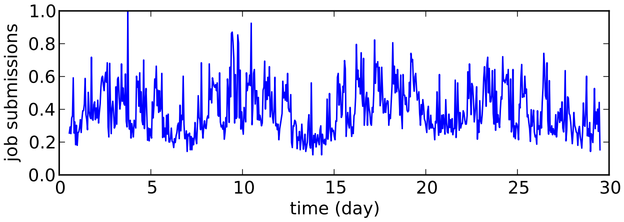


Figure 5: Job submissions in Google cluster

Capacity sharing: The SplitSRPT algorithm relies on being able to spilt/share the map capacity and shuffle capacity of the cluster. In practice, providing capacity sharing in the map phase is quite straightforward, since the map slots can simply be partitioned in order to create the desired sharing. Capacity sharing in the shuffle phase is more challenging, especially under the current Hadoop implementation, where the shuffle phase is attached to the reduce phase. As a result, the data transfer cannot start until reduce slots are allocated, while the actual reduce work cannot start until data transfer is done. One proposal for addressing this, and achieving capacity sharing, is to separate the shuffle phase from the actual reduce tasks so that the data transfer is not delayed by reducer allocation, e.g., as done in [39]. Though such approaches are not currently standard, they are certainly feasible.

5. NUMERICAL EXPERIMENTS

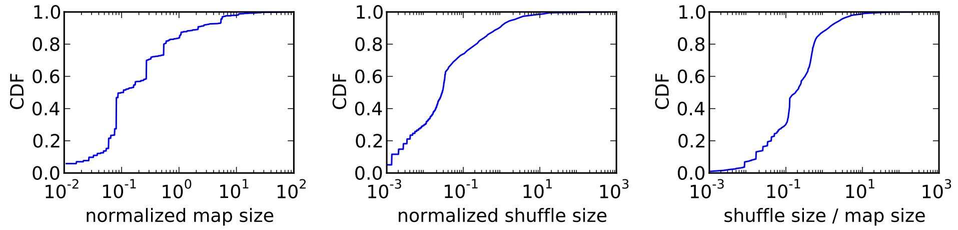
We have analyzed the worst-case performance guarantees for our algorithms in Section 4. In this section, our goal is to evaluate the algorithms under realistic workloads. To accomplish this, we perform numerical experiments using traces from a Google MapReduce cluster. We first describe the experimental setup in Section 5.1, and then we show the experiment results in Section 5.2. Finally, we study the question of unfairness in Section 5.3.

5.1 Experimental setup

In this section we introduce the workload trace for our experiments and the benchmark algorithms to compare with.

Workload information: The workloads we use for the experiments are cluster workload traces from Google [37]. The traces represent 29 days in May 2011 on a cluster of about 11k machines. The total size of the compressed trace is approximately 40GB. The trace recorded all the detailed information about job arrivals and resource usage for each task with timestamps. By filtering out the non-MapReduce jobs (e.g., jobs with single task, jobs with zero disk usage, etc.), we get about 80000 jobs submissions in 29 days, i.e., about 110 MapReduce jobs are submitted to the cluster per hour in average. The details information about job arrivals and job size statistics are described below.

The job submission rates (averaged per hour, normalizing the peak to 1.0, which is about 300 jobs/hour) are shown in Figure 5. We can see that there are strong diurnal traffic pattern and strong weekly traffic pattern for the cluster. Given such non-stationary traffic, it is very common or even crucial for the clusters to overprovision for the average load in order to handle the peak load during the busy hours or other traffic bursts. As an example, the average system load of 75% implies that the jobs submitted to the cluster during



(a) Map size distribution (b) Shuffle size distribution (c) Size ratio distribution

Figure 6: Job size distributions in the Google cluster

busy hours exceed twice of the cluster capacity, which is hardly usable in practice. To be realistic, in the experiments we focus on the scenarios that the average system load does not exceed 75%.

The map size distribution (with a normalized mean of 1) and the shuffle size distribution (with a normalized mean of 1) are shown in Figure 6(a) and 6(b) respectively. The size information for each job is collected by summing up the total resource usage of its tasks. Notice that the trace data does not include the intermediate data information for the shuffle phase, as a result we use the local disk usage (excluding the disk usage for the Google File System (GFS), which serves the input/output for the MapReduce jobs) to approximate the intermediate data size. Since all the resource usage is normalized in the original trace data, we also show the normalized results. From Figure 6(a) we can see that the map sizes spread in a wide range (standard deviation equal to 3.65), with maximum size more than 10^3 times of the minimum size. Another interesting observation is that there are a few jumps in the CDF figure, which indicate that there are a few job sizes making up of most workload for the cluster. Figure 6(b) shows that the shuffle sizes spread in a even wider range (standard deviation equal to 12.95), with maximum size more than 10^5 times of the minimum size, i.e., the intermediate data sizes are probably ranging in the order of a few MB to a few TB. The distribution of the ratio between map size and the shuffle size is shown in Figure 6(c) (standard deviation equal to 3.65). It can be seen that more than 80% of jobs are map-heavy (i.e., with map size bigger than shuffle size), while a small fraction of jobs having the shuffle size much bigger than the map size.

Simulation setup: We implement a discrete-event simulator for the overlapping tandem queue model, which supports many scheduling disciplines such as First Come First Serve, Weighted Processor Sharing, Limited Processor Sharing, MaxSRPT, and SplitSRPT.

To evaluate the performance of our algorithms, we choose two benchmarks to compare with. The first benchmark, which serves as the "baseline" policy, is to mimic the Fair Scheduler in Hadoop implementation. Fair Scheduler allocates the map slots to jobs in a fair manner, which is roughly a discrete version of Weighted Processor Sharing. Moreover, the Hadoop cluster can be configured to limit the maximum number of jobs running simultaneously and put the rest jobs in the queue. As a result, we use k-Limited Processor Sharing at the map station to mimic the Fair Scheduler policy, with $k = 100$, which is reasonable for a huge cluster. To approximate the default bandwidth sharing in the shuffle phase in Hadoop, we use Processor Sharing at the shuffle station. We call this benchmark "k-LPS" in the results.

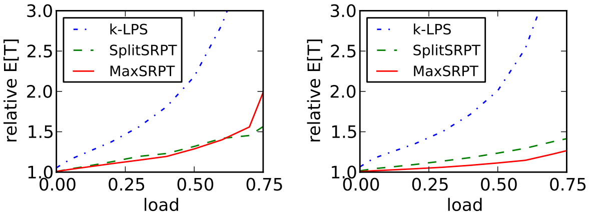
The second benchmark is not a particular scheduling policy but a lower bound of the mean response time for any policy. This lower bound is described below. Denote (x_i, y_i) the size of the i^{th} job, which is submitted at time r_i. Now consider a single server queue model A with the i^{th} job having size x_i released at r_i and a single server queue model B with the i^{th} job having size y_i released at r_i. Clearly SRPT is the best policy for both system A and B. Denote $\{t_j\}$ the time sequences where t_j is the j^{th} time instance that both system A and system B are idle, and there exists at least one arrival $r_i \in (t_{j-1}, t_j)$ $(t_0 = 0)$. Now consider the jobs arriving during time period (t_{j-1}, t_j) in the overlapping tandem queue model. Their total response time is not less than that under system A with corresponding job arrivals (which all arrived and completed during (t_{j-1}, t_j)), and also not less than that under system B with corresponding job arrivals (which all arrived and completed during (t_{j-1}, t_j)). Therefore, we choose the maximum of the total response time in system A and system B during the period (t_{j-1}, t_j) as a lower bound of the total response time for the jobs arriving during (t_{j-1}, t_j) in the overlapping tandem queue model. By summing up all the time periods, we get a lower bound of the total response time for our overlapping tandem queue model.

5.2 Experimental results

Using the workload and the benchmarks mentioned in Section 5.1, we can evaluate the performance of our algorithms. To make the comparison of algorithms more clear, we normalize the mean response time for different algorithm by the lower bound benchmark mentioned above and call it "relative $E[T]$", i.e. with relative $E[T]$ close to 1 implies the algorithm is near optimal, but near optimal algorithm may not necessarily have relative $E[T]$ equal to 1 since the lower bound can be loose.

To simulate the system at different load levels, we scale the cluster capacity such that the average system load varies in the range of $(0, 0.75)$. As we discussed above, 75% of average load already makes the job arrivals during the peak hours exceed twice of the cluster capacity due to the nonstationary traffic pattern.

The results are shown in Figure 7(a), where "k-LPS" represents the Fair Scheduler in Hadoop, and "MaxSRPT" and "SplitSRPT" represent our online algorithms. The first observation is that both our algorithms perform much better than the k-LPS policy, and both of them are not far from the lower bound. Given the fact that the lower bound comes from the single station models, which is very loose, we can conclude that both our algorithm are near-optimal for a wide range of loads. The second observation is that although our



(a)

(b)

Figure 7: Average response time under a Google trace. (a) shows mean response time under the original job sizes and (b) shows mean response time under the scaled job sizes.

two algorithms have very similar performance for a wide range of loads, SplitSRPT outperforms MaxSRPT when the load is high. This is probably because most of the jobs in the trace are very unbalanced and our analytical results show that SplitSRPT tends to have better performance in this scenario. To verify this conjecture, we scale the shuffle sizes of the jobs such that the size ratio for each job is the cube root of its original ratio, and then normalize the resulting shuffle size such that its mean is again 1. Intuitively, the scaled job sizes are more balanced than the original ones. The result of the experiment with the scaled job sizes is shown in Figure 7(b). Again, both of our algorithms outperform k-LPS and are near-optimal. As expected, MaxSRPT now has better performance than SplitSRPT given the more balanced job sizes.

5.3 What happens to large jobs?

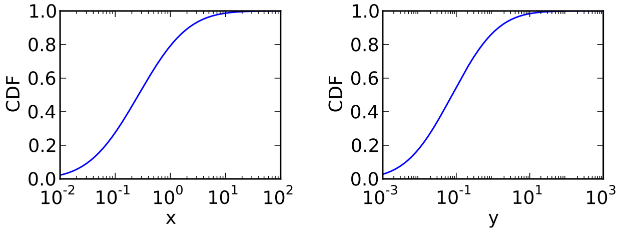
One common concern about SRPT-like scheduling policies is unfairness to large jobs. The worry is that because small jobs are given priority, large jobs may receive poor service.

In the context of a single server queue, the unfairness of SRPT has received significant attention, e.g., [11,12,34–36]. These studies have revealed that, in practical settings, large jobs actually receive as good (and sometimes better) performance under SRPT as they do under a fair scheduler like Processor Sharing (PS), which splits the service capacity evenly among all present jobs.

In this section, we study unfairness in the context of the overlapping tandem queue model, by comparing the MaxSRPT and SplitSRPT against the policy k-Limited Processor Sharing, k-LPS. We use k-LPS as the comparison for two reasons: (i) it models the "Fair Scheduler" in Hadoop, and (ii) it shares service capacity evenly among all the jobs in the system that are able to receive service.

To study unfairness, the typical approach, e.g., in [34, 35], is to use the conditional expected *slowdown* for a job of size z, $E[S(z)]$, which is the ratio between the expected response time for a job with processing time z divided by z. In this context, fair policies have expected $E[S(z)]$ that is independent of z, e.g., PS in M/GI/1 queue has $E[S(z)] = 1/(1-\rho)$, where ρ is the load.

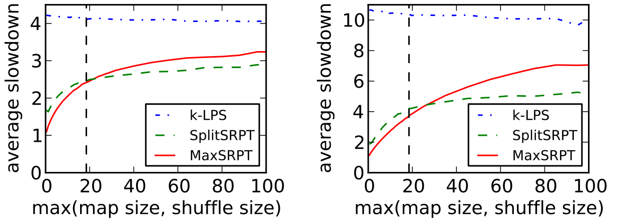
In the overlapping tandem queue model, we use the standalone job processing time to represent the size, i.e., $z = \max(x, y)$ for a job with size (x, y). Our baseline policy, k-LPS, will not have perfectly flat $E[S(z)]$ in the overlapping tandem queue; however $E[S(z)]$ will be nearly flat (slightly decreasing) in z for large k, and thus it is very nearly fair.



(a) log-normal map sizes

(b) log-normal shuffle sizes

Figure 8: Job sizes in the unfairness experiments



(a) load = 0.75

(b) load = 0.90

Figure 9: The average slowdown under synthetic workloads

Setup: Since there is a wide range of job sizes in the Google trace and the number of jobs in the trace is not statistical significant to show the average slowdown for different job sizes, we resort to the use of synthetic traces in this section. We use a variety of synthetic traces with different job size distributions such as log-normal, pareto, uniform and so on. Since the results are qualitatively similar, we only show the simulation results using log-normal job size distribution, with the parameter settings chosen to match those of the Google trace used previously.

Specifically, we use a log-normal distribution (mean equal to 1 and standard deviation equal to 3.65) to approximate the map sizes in Figure 6(a), and a log-normal distribution (mean equal to 1 and standard deviation equal to 12.95) to approximate the shuffle sizes in Figure 6(b). The distributions are shown in Figure 8. Note that the shuffle size of a job is usually not independent of its map size. Thus for each job, we generate its map size x_i using a log-normal distribution as shown in Figure 8(a), generate a ratio γ_i using another log-normal distribution (mean equal to 1.0 and standard deviation equal to 3.28) and let its shuffle size $y_i = x_i \cdot \gamma_i$. Then y_i is a log-normal distribution since it is the product of two log-normal random variables, with mean equal to 1 and standard deviation equal to 12.95.

To be statistically significant when examining the average slowdown for different job sizes, we run the our simulations with 5×10^7 job arrivals, with arrivals coming from a Poisson process. Then the jobs are classified into different buckets based on their stand-alone processing time, i.e., based on $L_i = \max(x_i, y_i)$. Finally the average slowdown for each bucket of jobs is calculated. Although job sizes can spread in a very wide range, 90% of jobs have L_i smaller than 3 and 99% of jobs have L_i smaller than 19. As a consequence, we show the average slowdown for $L_i \in [0, 100]$, where the jobs are classified into 400 buckets. Since the confidence intervals for each bucket are tight around the value shown, we do not draw them.

Results: Similarly to the results under the Google trace, the performance of MaxSRPT and SplitSRPT are much better than that of k-LPS under the synthetic traces. Specifically, under load of 0.75, the average response time for k-LPS is 6.50, the average response time for MaxSRPT is 3.32 and the average response time for SplitSRPT is 3.55; under load of 0.90, the average response time for k-LPS is 16.28, the average response time for MaxSRPT is 5.58 and the average response time for SplitSRPT is 5.66.

The conditional average slowdown $E[S(z)]$ under load of 0.75 and load of 0.90 are shown for each algorithm in Figure 9(a) and 9(b). As expected, the slowdown of k-LPS is nearly independent of job size, while the slowdown of MaxSRPT and SplitSRPT are increasing in the job size, which highlights the bias toward small job sizes in these policies. However, it can also be seen that this bias does not come at the expense of the large jobs. In particular, *all job sizes* have better expected performance under MaxSRPT and SplitSRPT than they do under k-LPS. This is a result of the fact that k-LPS does not keep the shuffle queue busy as often as MaxSRPT and SplitSRPT do, and thus MaxSRPT and SplitSRPT have a higher average service rate than k-LPS does. Further, remember that 90% of jobs are with $L_i < 3$ and 99% of jobs are with $L_i < 19$ (99^{th} percentile is indicated in Figure 9), and so Figure 9 highlights that most jobs have expected response time under our algorithms less than half of that under k-LPS. Another interesting observation is that SplitSRPT looks more fair than MaxSRPT. In both plots of Figure 9, the slowdowns for small jobs under SplitSRPT are bigger than that under MaxSRPT, but the slowdowns for big jobs under SplitSRPT are smaller than that under MaxSRPT.

6. CONCLUSION

This paper focuses on a challenging and important scheduling problem within MapReduce systems: how to schedule to minimize response time in the presence of phase overlapping of the map and shuffle phases.

A key contribution of this paper is the proposal of a simple, accurate, model for this phase overlapping. Within this model, this paper focuses on the design of algorithms for minimizing the average response time. We prove that this task is computationally hard in the worst case, even in the batch setting, but also show that it is possible to give approximately optimal algorithms in both the online and batch settings. The two algorithms we develop, MaxSRPT and SplitSRPT, are complementary to each other and perform quite well in practical settings.

The model presented here opens the door for an in depth study of how to schedule in the presence of phase overlapping. There are a wide variety of open questions remaining with respect to the design of algorithms to minimize response time. For example, are there algorithms with tighter worst-case guarantees? Is it possible for an online algorithm to be $O(n^{1/3})$-competitive? Is stochastic analysis feasible? Further, it is interesting and important to understand how to schedule in order to minimize other performance metrics, e.g., the number of deadlines met.

7. REFERENCES

[1] Fair Scheduler, `http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html`.

[2] Capacity Scheduler, `http://hadoop.apache.org/mapreduce/docs/r0.21.0/capacity_scheduler.html`.

[3] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: coordinated memory caching for parallel jobs. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012. USENIX Association.

[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 *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, pages 1–16, 2010.

[5] T. Bressoud and M. Kozuch. Cluster fault-tolerance: An experimental evaluation of checkpointing and MapReduce through simulation. In *Cluster Computing and Workshops, CLUSTER '09*, 2009.

[6] L. Cherkasova. Scheduling strategy to improve response time for web applications. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, London, UK, UK, 1998.

[7] J. Dai, J. Huang, S. Huang, B. Huang, and Y. Liu. Hitune: dataflow-based performance analysis for big data cloud. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 7–7, Berkeley, CA, USA, 2011.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[10] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, May, 1976.

[11] M. Gong and C. Williamson. Simulation evaluation of hybrid SRPT scheduling policies. In *Proc. of IEEE MASCOTS*, 2004.

[12] M. Gong and C. Williamson. Revisiting unfairness in web server scheduling. *Computer Networks*, 50(13):2183–2203, 2006.

[13] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979.

[14] Z. Guo, G. Fox, and M. Zhou. Investigation of data locality in MapReduce. *Cluster Computing and the Grid, IEEE International Symposium on*, pages 419–426, 2012.

[15] Hadoop. `http://hadoop.apache.org`.

[16] L. A. Hall. Approximability of flow shop scheduling. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 82–, Washington, DC, USA, 1995. IEEE Computer Society.

[17] M. Hammoud and M. Sakr. Locality-aware reduce task scheduling for MapReduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 570 –576, 2011.

[18] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2), May 2003.

[19] J. A. Hoogeveen and T. Kawaguchi. Minimizing total completion time in a two-machine flowshop: Analysis of special cases. *Mathematics of Operations Research*, 24(4):887–910, Nov. 1999.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*, 2007.

[21] M. Isard, V. Prabhakaran, J. Currey, U. Wieder,

K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*, pages 261–276. ACM, 2009.

[22] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production MapReduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103, Washington, DC, USA, 2010.

[23] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 110–119, New York, NY, USA, 1997. ACM.

[24] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós. On scheduling in map-reduce and flow-shops. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 289–298, 2011.

[25] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud. In *Proceedings of the International Conference on Supercomputing*, Seattle, WA, November 2011.

[26] D. Raz, H. Levy, and B. Avi-Itzhak. A resource-allocation queueing fairness measure. In *Proc. of ACM Sigmetrics-Performance*, 2004.

[27] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):pp. 687–690, 1968.

[28] R. Stewart and J. Singer. Comparing fork/join and MapReduce. In *Technical Report HW-MACS-TR-0096, Department of Computer Science, Heriot-Watt University.* Aug, 2012.

[29] J. Tan, X. Meng, and L. Zhang. Delay tails in MapReduce scheduling. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 5–16, New York, NY, USA, 2012. ACM.

[30] N. M. van Dijk. *Queueing networks and product forms - a systems approach.* Wiley-Interscience series in systems and optimization. Wiley, 1993.

[31] A. Verma, L. Cherkasova, and R. Campbell. Two sides of a coin: Optimizing the schedule of MapReduce jobs to minimize their makespan and improve cluster performance. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE 20th International Symposium on*, 2012.

[32] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for MapReduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing (ICAC)*, 2011.

[33] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, and U. Dayal. Modeling the performance of the Hadoop online prototype. In *Proc. of Computer Architecture and High Performance Computing*, pages 152–159, 2011.

[34] A. Wierman. Fairness and classifications. *ACM SIGMETRICS Performance Evaluation Review*, 34(4):4–12, 2007.

[35] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. In *Proc. of ACM SIGMETRICS*, 2003.

[36] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to higher moments of conditional response time. In *Proc. of ACM SIGMETRICS*, 2005.

[37] J. Wilkes. More Google cluster data. Google research blog, Nov. 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[38] R. W. Wolff. *Stochastic Modeling and Theory of Queues.* Prentice Hall, 1997.

[39] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user MapReduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, April 2009.

[40] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[41] H. C. Zhao, C. H. Xia, Z. Liu, and D. Towsley. A unified modeling framework for distributed resource allocation of general fork and join processing networks. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2010.

APPENDIX

A. HARDNESS PROOFS

Proof of Theorem 1. Given any instance of the NMTS problem, note that we are free to add a constant to each g_i and h_i without changing the problem. Thus, we can assume that $g_i > 3\sum_{j=1}^{q} f_j$ for $i = 1, \ldots, q$.

Now let us construct an input for our scheduling problem based on the given instance of NMTS problem. Let A be a positive integer which satisfies $A \geq 2(\sum_{i=1}^{q} g_i + 2\sum_{i=1}^{q} f_i)\sum_{i=1}^{q} g_i$. There are q Type-F jobs in the system with job sizes $(A, A+f_i)$ (denoted by F_i), q Type-G jobs with job sizes $(A, A+g_i)$ (denoted by G_i) and q Type-H jobs with job sizes $(A, A-h_i)$ (denoted by H_i). There are also two Type-E jobs (denoted by E_1 and E_2) with size $(A, A+M)$ where $M \geq 9qA$. The threshold we are interested in is $y = (9q^2/2+3q/2)A + 2\sum_{i=1}^{q} f_i + \sum_{i=1}^{q} g_i + 6qA + 3M + 3A$.

If there does not exist a schedule with cost no more than y, then we can conclude that the answer to the corresponding NMTS problem is no. Otherwise, suppose there exist permutations π and σ in the NMTS problem, then the scheduling order $F_{\pi(1)}$, $G_{\sigma(1)}$, H_1, $F_{\pi(2)}, \ldots, H_q$, E_1, E_2 gives us total response time of

$$\begin{aligned} T &= \textstyle\sum_{i=1}^{q}(9(i-1)A + (A + f_{\pi(i)}) + (2A + f_{\pi(i)} + g_{\sigma(i)}) \\ &\quad +3A) + (3qA + A + M) + (3qA + 2A + 2M) \\ &= y \end{aligned}$$

and thus we get a contradiction.

If there exists a scheduling with cost no more than y, we show in the following that the answer to the corresponding NMTS problem is yes.

First, we can argue that the two Type-E jobs are completed last since, otherwise, if another job is completed after a Type-E job then the total response time is greater than $(9q^2/2 - 3/2q)A + 4M + 4A$, which is greater than y.

Denote t_k the completion time of the kth completed job under this schedule. Obviously we always have $t_k \geq kA$ for $k = 1, \ldots, 3q$ and $t_k \geq 3qA + (M + A)(k - 3q)$ for $k = 3q+1, 3q+2$ no matter the schedule. Thus, a trivial lower bound for the total response time is $y_0 = (9q^2/2 + 3q/2)A + 6qA + 3M + 3A$. Denote $y' = y - y_0 = 2\sum_i f_i + \sum_i g_i$.

Notice that, although jobs may preempt each other or share the capacity of station, we always have $t_k \leq kA + y'$ since otherwise the total response time is above y. Now let us consider the completion time of a Type-G job G_i. Assume it completed at t_j. Notice that no more than y' units of its map workload can be done before t_{j-1}, otherwise $t_{j-1} > (j-1)A + y'$ and thus the total response time is above y. Therefore, at least $A - y'$ units of G_i's map workload is finished in $[t_{j-1}, t_j]$. Thus, the amount of G_i's shuffle workload done in $[t_{j-1}, t_j]$ is at least $(1 - y'/A)g_i$ more than its map workload done in the same period. Notice that its map workload is finished no earlier than jA; thus $t_j \geq jA + (1-y'/A)g_i$. Therefore, all Type-G jobs contribute at least extra $(1-y'/A)\sum_{k=1}^{q} g_k$ to the total response time. Now, suppose that the job completed at t_{j+1} is not a Type-H job. Then, we know that the amount of its shuffle workload done in $[t_{j-1}, t_{j+1}]$ is also at least as much as its map workload done in the same period. Denote the amount of its shuffle workload done in $[t_{j-1}, t_{j+1}]$ minus the amount of its map workload done in the same period by s. Now let us focus on the total workload of G_i and this job done in $[t_{j-1}, t_{j+1}]$. The total shuffle workload done is at least $(1 - y'/A)g_i + s$ more than the map workload done during this period. Notice that the total map workload is done no earlier than $(j+1)A$. Thus $t_{j+1} \geq (j+1)A + (1-y'/A)g_i + s$, i.e., finishing this job after G_i will delay its response time at least an extra of $(1 - y'/A)g_i$ (without double counting the delay coming from s). The total response time is then not less than $y_0 + (1 - y'/A)\sum_{k=1}^{q} g_k + (1 - y'/A)g_i$, which is greater than y (since $A > 2y' \sum_{k=1}^{q} g_k / \sum_{k=1}^{q} f_k$) and we get a contradiction. Therefore, each Type-G job is followed by a Type-H job.

Now let us consider Type-F jobs. Obviously the job completed right after a Type-F job can not be a Type-H job since each Type-H job is completed right after a Type-G job. Based on the same argument as above for the Type-G jobs, all Type-F jobs and the jobs completed right after contribute at least extra $2(1-y'/A)\sum_{k=1}^{q} f_k$ to the total response time. Suppose the job completed after the job completed right after a Type-F job F_i is not a Type-H job, then this job contributes another extra response time $(1-g'/A)f_i$ to the total response time. In this case, the total response time is not less than $y_0 + (1 - y'/A)\sum_{k=1}^{q} g_k + 2(1 - y'/A)\sum_{k=1}^{q} f_k + (1-y'/A)f_i$, which is greater than y (since $A \geq 2y' \sum_{k=1}^{q} g_k$) and we get a contradiction.

Therefore, the only possible completion order is $F_{\pi(1)}$, $G_{\sigma(1)}$, $H_{\phi(1)}$, $F_{\pi(2)}$,..., $H_{\phi(q)}$, E_1, E_2, for some permutations π, σ and ϕ. The total response time is at least $y_0 + (1-y'/A)\sum_{k=1}^{q} g_k + 2(1-y'/A)\sum_{k=1}^{q} f_k$. Now suppose there exist $F_{\pi(i)}, G_{\sigma(i)}, H_{\phi(i)}$ such that $f_{\pi(i)} + g_{\sigma(i)} > h_{\phi(i)}$, i.e., $f_{\pi(i)} + g_{\sigma(i)} \geq h_{\phi(i)} + 1$ since $f_{\pi(i)}$, $g_{\sigma(i)}$, $h_{\phi(i)}$ are all positive integers. Denote their completion time t_j, t_{j+1}, t_{j+2}. Let us focus on the period $[t_{j-1}, t_{j+2}]$. Clearly, no less than $A - y'$ map workload for each job of $F_{\pi(i)}$, $G_{\sigma(i)}$ and $H_{\phi(i)}$ are done in this period, and thus the total shuffle workload of $F_{\pi(i)}$, $G_{\sigma(i)}$ and $H_{\phi(i)}$ done in this period is at least $(1 - y'/A)(f_{\pi(i)} + g_{\sigma(i)} - h_{\phi(i)}) \geq 1 - y'/A$ more than their total map workload done in the same period. Therefore, the completion time of $H_{\phi(i)}$ is $t_{j+2} \geq (j + 2)A + (1 - y'/A)$. Now the total response time is not less than $y_0 + (1 - y'/A)\sum_i g_i + 2(1 - y'/A)\sum_i f_i + (1 - y'/A)$, which is not less than y (since $A \geq 2y' \sum_{k=1}^{q} g_k$) and we get a contradiction. Thus $f_{\pi(i)} + g_{\sigma(i)} \leq h_{\phi(i)}$ for $i = 1, \ldots, q$. Note $\sum_{k=1}^{q} f_k + \sum_{k=1}^{q} g_k = \sum_{k=1}^{q} h_k$ by the definition of NMTS, we have $f_{\pi(i)} + g_{\sigma(i)} = h_{\phi(i)}$ for $i = 1, \ldots, q$.

In summary, the only way to have the total response time no more than y is to have the completion order $F_{\pi(1)}$, $G_{\sigma(1)}$, $H_{\phi(1)}$, $F_{\pi(2)}$,..., $H_{\phi(q)}$, E_1, E_2, for some permutations π, σ and ϕ. Moreover, the permutations satisfy $f_{\pi(i)} + g_{\sigma(i)} = h_{\phi(i)}$ for $i = 1, \ldots, q$. This implies that the original NMTS problem has a solution, and thus finishes the reduction. □

PROOF OF THEOREM 2. In order to prove the result we construct an instance that guarantees that any algorithm will have a large average response time.

Denote $k = \lfloor n^{1/3} \rfloor$. Consider an instance where k^2 type-1 jobs and k type-2 jobs arrive at $t = 0$. Type-1 jobs have size $(1, 1/2)$ and type-2 jobs have size $(k, 3k/2)$, where the first value represents the map size and the second value represents the shuffle size.

Deterministic algorithms: Under a given deterministic algorithm, we consider two cases depending on how much map workload of type-2 jobs has been finished by time $t = k^2$.

Case (1): At least $k^2/5$ units of map workload of type-2 jobs has been finished. In this case, there are at least $k^2/5$ units of map workload of type-1 jobs unfinished at time $t = k^2$. Consider the future arrival of type-1 jobs released one by one at $t = k^2 + j$ $(j = 0, 1, \ldots, k^3)$, then the number of jobs in the system is not less than $k^2/5 + 1$ during the period $[k^2, k^2 + k^3]$. Note that the total response time is the integral over time of the number of jobs in the system. Thus, the total response time during the period $[k^2, k^2 + k^3]$ is $T > (k^2/5 + 1)k^3$. Now consider an alternative algorithm that finishes all type-1 jobs by $t = k^2$. Then there are only k (type-2) jobs unfinished at time $t = k^2$. With the same arrival of jobs during the period $[k^2, k^2+k^3]$, the total response time for all jobs is $T' < (k^2 + k) \cdot k^2 + (k+1) \cdot k^3 + k \cdot 3k^2/2$. Therefore, $T/T' = \Omega(k)$.

Case (2): Less than $k^2/5$ units of map workload of type-2 jobs has been finished. Then there are at least $6k^2/5$ units of shuffle workload of type-2 jobs unfinished at $t = k^2$. Thus, there will be at least $k^2/5$ units of shuffle workload of type-2 jobs unfinished at $t = 2k^2$. Consider the future arrival of type-2 jobs released one by one at $t = 2k^2 + j \cdot 3k/2$ $(j = 0, 1, \ldots, k^3)$, then the number of jobs in the system is not less than $(k^2/5)/(3k/2) + 1$ during the period $[2k^2, 2k^2 + 3k^4/2]$. Thus, the total response time is $T > (2k/15 + 1) \cdot 3k^4/2$. Now, consider an alternative algorithm that schedules type-2 jobs one by one and then type-1 jobs one by one in $[0, 2k^2]$. Then, all $(k^2 + k)$ jobs are done by time $t = 2k^2$. With the same arrival of jobs during the period $[2k^2, 2k^2 + 3k^4/2]$, the total response time for all jobs is $T' < (k^2 + k) \cdot 2k^2 + 3k^4/2$. Therefore, $S/S' = \Omega(k)$.

Randomized algorithms: For a randomized algorithm, with k^2 type-1 jobs and k type-2 jobs arriving at $t = 0$, there are again two cases we consider. Case (1): At least $k^2/5$ units of map workload of type-2 jobs has been finished by $t = k^2$. Case (2): Less than $k^2/5$ units of map workload of type-2 jobs has been finished by $t = k^2$.

If Case (1) happens with probability not less than $1/2$, then we consider the arrival of type-1 jobs released one by one at $t = k^2 + j$ $(j = 0, 1, \ldots, k^3)$. For this input instance, the expected total response time of the randomized algorithm is $\Omega(k^5)$, while the optimal offline solution having total response time $O(k^4)$.

The same argument holds when Case (2) happens with probability not less than $1/2$. Therefore, the competitive ratio for randomized algorithms is also $\Omega(k)$. □