Joint Scheduling of Overlapping MapReduce Phases: Pair Jobs for Optimization

Huanyang Zheng and Jie Wu, *Fellow, IEEE*

Abstract—MapReduce includes three phases of map, shuffle, and reduce. Since the map phase is CPU-intensive and the shuffle phase is I/O-intensive, these phases can be conducted in parallel. This paper studies a joint scheduling optimization of overlapping map and shuffle phases to minimize the average job makespan. New concepts of the strong pair and the weak pair are introduced. Two jobs are defined as a strong pair if the shuffle and map workloads of one job equal the map and shuffle workloads of the other job, respectively. Two jobs are defined as a weak pair if their total map workloads equal their total shuffle workloads. We prove that if the entire set of jobs can be decomposed to strong pairs of jobs, then the optimal schedule can pairwisely execute jobs that can form a strong pair. Following the above intuition, several offline and online scheduling policies are proposed. Extensions are made based on weak pairs. Real data-driven experiments validate the efficiency and effectiveness of the proposed policies.

Index Terms—MapReduce framework, map and shuffle phases, joint scheduling, makespan optimization.

1 Introduction

MapReduce is a well-known programming framework used to process the ever-growing amount of data collected by modern instruments, such as the Large Hadron Collider and next-generation gene sequencers. Although MapReduce has been widely adopted in a number of data centers, more improvements are still needed to meet the huge demands of big data computing. In the current MapReduce framework, each job consists of three dependent phases: *map*, *shuffle*, and *reduce*. The map and reduce phases generally deal with a large amount of data computations, while the shuffle phase transfers the data among different MapReduce workers. In terms of the resource demand, the map and reduce phases are CPU-intensive, while the shuffle phase is I/O-intensive.

Currently, most state-of-the-art research on MapReduce optimizations focuses on the map and reduce phases. However, the shuffle phase also plays a very important role in transferring the data from map workers to reduce workers. It has a significant impact on the average job makespan, especially when the data is big. Moreover, Chen et al. [1] reported that jobs processed by the Facebook MapReduce cluster are shuffle-heavy. Consequently, this paper studies a joint scheduling optimization of map and shuffle phases to *minimize the average job makespan* (the time span from the job arrival to the completion of the shuffle phase). The reduce phase is not jointly optimized since its workload is relatively light. According to Zaharia et al. [2], only 7% of jobs in a production MapReduce cluster are reduce-heavy.

MapReduce usually involves a lot of nodes across the network. This paper focuses on single node optimizations for MapReduce. Our key observation is that the map and shuffle phases have different resource demands for a single node in MapReduce. Since the map phase is CPU-intensive and the shuffle phase is I/O-intensive, they can potentially be conducted in parallel to minimize the average job makespan. The key challenge comes from the fact that the map and shuffle phases cannot be fully parallelized due to their *dependency relationship*. The shuffle phase of a job must start later than its map phase, and cannot finish earlier than its map phase. This is because the shuffle phase may wait to transfer the data emitted by the map phase. An example includes the WordCount [3], in which the map workers emit key-value pairs at a certain rate to be shuffled to the reduce workers. If the map workload of a job is larger than its shuffle workload, the I/O resource may be underutilized, leading to a non-optimal job schedule. In addition, this paper considers that the job workload to be fixed as a prior (map and shuffle workloads are not symbiotic, and applications such as SecondarySort are not considered).

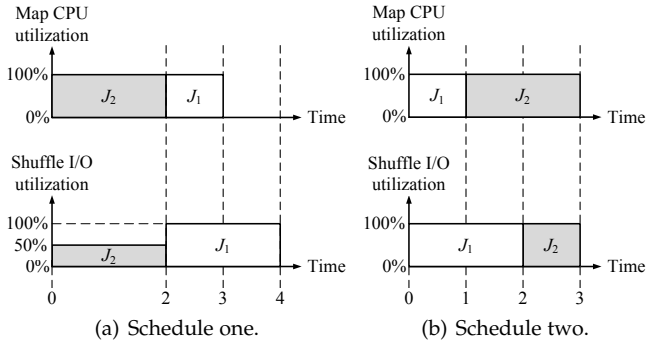


Fig. 1. An example for the joint scheduling of overlapping phases.

To illustrate the above motivation more clearly, an example is shown in Fig. 1, which involves two jobs of J_1 and J_2. J_1 is shuffle-heavy and J_2 is map-heavy. Assuming that the resources are fully utilized, the map and shuffle phases of J_1 take 1 and 2 time slots, respectively. The resource demand of J_2 is the opposite of that of J_1 (1 time slot for the shuffle phase and 2 time slots for the map phase). As shown in Fig. 1(a), schedule one executes J_2 first, leading

• *H. Zheng and J. Wu are with Center for Networked Computing, Temple University, Philadelphia, PA 19122, USA.*
E-mail: {huanyang.zheng, jiewu}@temple.edu

Manuscript received April 19, 2017; revised August 26, 2017.

to an underutilization of the I/O resource. This is because J_2's shuffle phase needs to wait to transfer the data emitted by its map phase (suppose a constant data emission rate). Consequently, schedule one takes 4 time slots to finish all the jobs. As shown in Fig. 1(b), schedule two is a better scheme. It executes J_1 first and only takes 3 time slots to finish all the jobs. It can be seen that, in order to maximally utilize the I/O resource, the shuffle-heavy job should be executed earlier than the map-heavy job.

New concepts of the strong job pair and the weak job pair are introduced to address the above problem. Two jobs are called a *strong pair* if the shuffle and map workloads of one job equal the map and shuffle workloads of the other job, respectively. Two jobs are called a *weak pair* if their total map workloads equal their total shuffle workloads. We prove that if the entire set of jobs can be decomposed to strong pairs of jobs, then the optimal schedule is to pairwisely execute jobs that can form a strong pair. Several offline and online scheduling algorithms are proposed to minimize the average job makespan. They first group jobs according to job workloads, and then, execute jobs within each group through a pairwise manner. Extensions are made based on weak pairs.

The remainder of this paper is organized as follows. Section 2 surveys the related works. Section 3 describes the model and formulates the problem. Sections 4 and 5 study the offline job scheduling with strong and weak job pairs, respectively. Online scheduling is extended. Section 6 includes extensive real data-driven experiments. Finally, Section 7 concludes this paper and discusses future directions.

2 Related Work

Extensive studies on the MapReduce scheduler have been conducted over the past few years. An example includes the delay scheduling [4], which postpones the task scheduling and ameliorates the locality degradation in the Hadoop scheduler. Another example is the ARIA [5], which allocates appropriate amounts of resources to each MapReduce job to meet service level objectives. Zhang et al. [6] improved ARIA by estimating the amount of resources required to complete a program. Wolf et al. [7] proposed a framework to optimize different scheduling metrics, based on a performance model, with respect to the job execution time. Tang et al. [8] proposed a scheduling policy that dynamically determines the start time of each reduce task according to its job context. Mantri [9] can mitigate the impact of outliers. It monitors task executions with real-time outlier estimations, then takes reactions, such as restarting and terminating specified outliers. Tarazu [10] was a communication-aware scheme, which schedules predictive load-balancing MapReduce jobs to reduce the network traffic within heterogeneous Hadoop clusters. Quincy [11] achieved a balanced tradeoff between the job fairness and the data locality through a min-cost flow method and a preemption mechanism. Amoeba [12] supported lightweight elastic tasks that can release the CPU resources without losing I/O computations. Moreover, multi-resource (CPU and I/O) packing problems were also investigated for MapReduce schedulers [13–15]. For example, Graphene [13] was designed to schedule jobs that have complex dependency structures and heterogeneous resource demands. Graphene focused on the long-running tasks and those with tough-to-pack resource demands. These troublesome tasks can be scheduled in advance of the remaining tasks without violating the dependency constraints. PRISM [14] divided tasks into several phases, where each phase has a constant resource usage profile, and performs scheduling at the phase level. The importance of phase-level scheduling was demonstrated by the resource usage variability within the lifetime of a task using a wide-range of MapReduce jobs. A phase-level scheduling algorithm was also introduced to improve execution parallelism and resource utilization. Verma et al. [15] developed a method to break the barrier between the Map and Reduce stages in MapReduce, in order to improve the efficiency. A barrier-less MapReduce framework was designed to obtain the equivalent generality and retain ease of programming. However, the above works focus on the resource scheduling for map and reduce phases. The overlapping shuffle phase is not jointly optimized.

In 2013, Lin et al. [16] proposed a landmark model for the overlapping map and shuffle phases in MapReduce. They proved that the problem of minimizing the average job makespan is NP-hard in the offline scenario and APX-hard in the online scenario. Therefore, no online scheduling policy can guarantee a constant approximation ratio with respect to the optimal scheduling policy. However, Lin's scheduling policy may not be efficient, since the optimal pattern is under-explored. We show that optimal results can be obtained by pairing map-heavy jobs and shuffle-heavy jobs under load-balancing offline scheduling scenarios. Li et al. [17] considered a model with overlapping shuffle and reduce phases, utilizing the data locality to minimize the time for the shuffle phase. However, Li's scheduling policy does not guarantee an approximation ratio over time. This paper is also related to Wang's research [18], where the shuffle phase is reconfigurable to dynamically coordinate the map and reduce phases. A mathematical model was proposed to judge the computing complexities with different operating orders within the map-side shuffle, so that a faster execution can be achieved through reconfiguring the order of sorting and grouping. Some sampled features during the shuffle stage were collected to support the evaluation of the computing complexities of each operating order. By contrast, this paper optimizes the MapReduce with a fixed shuffle workload. In addition, our scheduling policy is similar to OMO [19], which aimed to optimize the overlap between the map and reduce phases. OMO is based on the lazy start of reduce tasks and the batch finish of map tasks, which catch the characteristics of the overlap and achieve a good alignment of the two phases.

Our problem is a variation of the flow shop scheduling problem [20], which is a class of scheduling problems with a set of machines. Each job is processed on this set of machines in compliance with the given processing orders. A continuous flow of jobs is scheduled with the objective of a minimum completion time or waiting time. Flow shop scheduling is a special case of job shop scheduling [21], in which there is a strict order of machines for each job to be processed [22]. Our problem is similar to a flow shop scheduling problem with two machines: one machine represents the map phase, while the other machine represents the shuffle phase. The difference is that our problem minimizes

the average job makespan, while the flow shop scheduling problem minimizes the completion time or the waiting time. As a result, our problem prefers to schedule jobs with lighter workloads before jobs with heavier workloads, but the flow shop scheduling problem does not have such a preference. It has been proven that the flow shop scheduling problem with only two machines can be optimally solved in a polynomial time, but the flow shop scheduling problem with more than two machines is NP-hard [20]. There are some extensions of the flow shop scheduling problem. Wang et al. [23] proposed an effective distribution algorithm to solve the distributed flow shop scheduling problem. The earliest completion factory rule was employed based on an encoding that generates feasible schedules and calculates the schedule objective value. A probability model was built for describing the probability distribution of the solution space. Marichelvam et al. [24] presented a cuckoo search meta-heuristic algorithm to minimize the makespan for the flow shop scheduling problem. A constructive heuristic was incorporated to obtain the near-optimal solutions rapidly.

3 Model and Problem Formulation

This paper focuses on a MapReduce framework with overlapping map and shuffle phases. In MapReduce, map workers continuously emit processed data (at a constant rate), which are in turn shuffled to reduce workers. We consider that map and shuffle phases mainly take CPU and I/O resources, respectively. Hence, they may be conducted in parallel. However, the shuffle phase is dependent on the map phase. This is because the shuffle phase may wait to transfer the data emitted by the map phase. If the data transfer rate of the shuffle phase is higher than the data emission rate of the map phase, then the shuffle phase has to wait for the data emission. As a result, the shuffle phase of a job must start later than its map phase and cannot finish earlier than its map phase. The reduce phase is not jointly optimized, since its workload is light [2].

We study both *offline* and *online* scenarios with n jobs in total. The offline scenario means that all jobs arrive at the system *at the start time*, waiting to be scheduled (job information is pre-known). The online scenario means that the scheduler only obtains the workload information of a job upon its arrival, which may not be the start time. Let $J = \{J_1, J_2, ..., J_n\}$ denote the set of jobs, where J_i is the ith job. Let t_i^m and t_i^s denote the map and shuffle workloads of J_i, respectively. The workload of a job is its execution time under fully-utilized resources. A MapReduce job may include multiple parallel subtasks on different machines. In such an event, its workload is the sum among different subtasks. The CPU resource is always fully utilized. In contrast, the I/O resource may be underutilized due to the dependency relationship between the map and shuffle phases. The actual shuffle time is considered to be reversely proportional to the I/O utilization for model simplicity. For example, when the I/O utilization is 25%, the shuffle time is quadrupled. Note that this assumption can be improved. One reason is that MapReduce framework manipulates the shuffle with patch pattern, i.e., shuffle data is not delivered to network until a local data buffer is fully filled. As a result, shuffle can present burst of data at I/O ports. Other reasons can be overhead and and machine performance variance. However, when the shuffle workload is large enough with respect to the local buffer, this assumption becomes solid in terms of approximating the average performance [19].

TABLE 1
Notations.

J_i and n	J_i is the ith job and n is the number of jobs
J	set of jobs, $J = \{J_1, J_2, ..., J_n\}$
t_i^m and t_i^s	the map and shuffle workloads of J_i
S	a schedule for J
G_i	the ith job group
$\Delta_{i,j}$	the job priority difference between J_i and J_j
α	a weight parameter
k	a parameter to determine number of job groups

We have the following definitions:

Definition 1. The job of J_i is said to be balanced if and only if $t_i^m = t_i^s$. If $t_i^m > t_i^s$, J_i is map-heavy. On the other hand, if $t_i^m < t_i^s$, J_i is shuffle-heavy.

Definition 2. The makespan of a job is the time span from its arrival to its shuffle phase completion, including its waiting time before the job execution.

The objective of this paper is to *minimize the average job makespan* through jointly scheduling overlapping map and shuffle phases. We do not minimize the latest job completion time (or other objectives) since these objectives have been well-studied in the flow shop scheduling field [20]. We assume that the MapReduce has a *centralized scheduler*, which abstracts the job schedule as a *sequential order*. The scheduler executes the next job, only if the MapReduce cluster has sufficient machines with idle CPU resources. This is because the next job may require the CPU resources of multiple machines to start its map phase. Our problem is NP-hard and APX-hard in the offline and online scenarios, respectively [16]. Therefore, this paper studies some special cases to design effective heuristics for our problem.

Note that a job may not execute immediately after its arrival, since it may be scheduled to wait for other jobs. To minimize the average job makespan, we prefer to execute jobs with lighter workloads earlier. This is because the smaller jobs can finish earlier. This preference can introduce an unfair policy that many small tasks could block the issue of large tasks for a long interval (algorithm performances can be degraded if considering the fairness issue). However, this paper does not explore the fairness problem for simplicity. The key challenge comes from the dependency relationship between the map and shuffle phases, which may lead to I/O underutilization (and thus, a non-optimal schedule). As a result, the optimal schedule may not be simply ranking jobs by their workloads. The following section will explore some insights. Finally, all notations are shown in Table 1.

4 Offline Scheduling with Strong Job Pair

4.1 Strong Job Pair and Its Optimality

To obtain more insights on the offline scheduling, we start with a special case of J, based on the following definition:

Definition 3. Two jobs, J_i and J_j, are called a strong pair if $t_i^m = t_j^s$ and $t_i^s = t_j^m$.

If two jobs can form a strong pair, then their map and shuffle workloads are exactly opposite to each other, meaning that

Algorithm 1 Pair-based Scheduling Policy

Input: The job set, J, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.

1: Initialize an array, S, to denote the job execution order;
2: Put all jobs into the order array of S;
3: Sort all jobs in S according to $\max(t_i^m, t_i^s)$;
4: **for** each subset of jobs with the same $\max(t_i^m, t_i^s)$ **do**
5: Reorder jobs by iteratively taking out a pair of jobs of $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$;
6: **return** the order array of S as the schedule;

they can be executed together to avoid I/O underutilization. This is a special case that can result in the optimal offline schedule, as shown in the following theorem [25]:

Theorem 1. If J can be decomposed to strong pairs of jobs, then jobs that can form a strong pair are pairwisely executed in the optimal offline schedule for J. For each strong job pair, the shuffle-heavy job is executed before the map-heavy job.

The proof of Theorem 1 is described in [25]. It means that we can avoid I/O underutilization by pairwisely executing jobs that can form a strong pair. This idea can be extended by organizing a bundle of jobs (such as a 3-tuple of jobs) as a basic scheduling unit. However, such an extension may bring a higher scheduling complexity and may post a higher optimality prerequisite on the workload distributions of jobs. Therefore, we use a pair of jobs (rather than a 3-tuple of jobs) as the basic scheduling unit.

4.2 Pair-based Scheduling and Discretization

Our first idea is to schedule jobs based on their workloads and try to pair jobs that have the same workloads based on Theorem 1. Consequently, Algorithm 1 is proposed, which has two stages. The first stage (lines 1 to 3) is based on Lin's MaxSRPT algorithm [16], where jobs are sorted according to $\max(t_i^m, t_i^s)$. Note that $\max(t_i^m, t_i^s)$ represents the dominant workload of J_i. Jobs with lighter workloads should be executed earlier, since small jobs could finish earlier to minimize the average job makespan. The second stage (lines 4 and 5) is our novel contribution based on Theorem 1. Jobs are iteratively paired according to their map and shuffle workload differences. We prioritize jobs with smaller workloads (the first stage) over jobs with better pairs (the second stage), since the former one generally rules the latter one (as verified in experiments). The time complexity of Algorithm 1 is $O(n \log n)$, and n is the number of jobs. This time complexity results from the sorting procedure in Algorithm 1 (lines 3 and 5).

As shown in [25], Algorithm 1 works well *when only a small portion of jobs can be paired*.

Theorem 2. Algorithm 1 is optimal when all jobs in J are simultaneously map-heavy, balanced, or shuffle-heavy.

4.3 Couple-based Scheduling and Generalization

We find that Algorithm 1 fails to work well when a large portion of jobs can be paired. Therefore, Algorithm 2 is proposed to address the above issue. Similar to Algorithm 1,

Algorithm 2 Couple-based Scheduling Policy

Input: The job set, J, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.

1: Initialize an array, S, to denote the job execution order;
2: Put all jobs into the order array of S;
3: Sort all jobs in S according to $t_i^m + t_i^s$;
4: **for** each subset of jobs with the same $t_i^m + t_i^s$ **do**
5: Reorder jobs by iteratively taking out a pair of jobs of $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$;
6: **return** the order array of S as the schedule;

Algorithm 3 Generalized Scheduling Policy

Input: The job set, J, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.

1: Initialize an array, S, to denote the job execution order;
2: Put all jobs into the order array of S;
3: Set J_i's priority as $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$;
4: Sort all jobs in S according to their priorities;
5: **for** each subset of jobs with the same priority **do**
6: Reorder jobs by iteratively taking out a pair of jobs of $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$;
7: **return** the order array of S as the schedule;

Algorithm 2 also has two stages. In the first stage (lines 1 to 3), all jobs are sorted according to their total map and shuffle workloads, i.e., $t_i^m + t_i^s$. Its intuition is similar to that of Algorithm 1: jobs with lighter workloads should be executed earlier, since the smaller jobs can finish earlier to minimize the average job makespan. The key difference is that jobs are sorted by total map and shuffle workloads in Algorithm 2, instead of dominant workloads in Algorithm 1. The second stage of Algorithm 2 (lines 4 and 5) is identical to Algorithm 1, where jobs are iteratively paired based on their map and shuffle workload differences. The time complexity of Algorithm 2 remains $O(n \log n)$1.

As shown in [25], Algorithm 2 works well *when a large portion of jobs can be paired*.

Theorem 3. Algorithm 2 is optimal when J can be decomposed to strong pairs of jobs.

Moreover, we have the following corollary:

Corollary 1. Algorithms 1 and 2 are equivalent and optimal when all jobs in J are simultaneously balanced.

While Algorithm 1 works well when only a small portion of jobs can be paired, Algorithm 2 works well when a large portion of jobs can be paired. They are equivalent and optimal when all jobs are balanced. To balance this tradeoff, Algorithm 3 is proposed to combine Algorithms 1 and 2. It uses $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$ as J_i's priority, then sorts all jobs according to their priorities. α serves as a weight parameter that satisfies $0 \leq \alpha \leq 1$.

4.4 Group-based Policy

Previous subsections introduced Algorithms 1, 2, and 3 to schedule jobs with a discretization process, which controls the granularity of the job priority. Jobs with similar priorities are grouped for the pairing process. The discretization

Algorithm 4 Group-based Scheduling Policy

Input: The job set, J, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.

1: Initialize an array, S, to denote the job execution order;
2: Put all jobs into the order array of S;
3: Set J_i's priority as $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$;
4: Sort all jobs in S according to their priorities;
5: Divide jobs into k groups by dynamic programming:
   Initialize a two-dimensional array of OPT;
   Initialize $OPT_{j,l} = 0$ when $j = 0$ or $l = 0$;
   Compute $OPT_{j,l} = \min_{l \le i \le j}\{OPT_{i-1,l-1} + \Delta_{i,j}\}$;
   Trace back the optimal job grouping through index i;
6: **for** each group of jobs **do**
7: Reorder jobs by iteratively taking out a pair of jobs of $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$;
8: **return** the order array of S as the schedule;

process is essentially a grouping (or clustering) procedure, and thus, it could be replaced by other grouping methods. This subsection presents a pairwise scheduling policy that groups jobs through a dynamic programming approach. The grouping goal is to divide jobs into k groups, such that the Sum of the Maximum Job Priority Difference within each group (SMJPD) is minimized. Let $G_1, ..., G_k$ denote the k job groups. SMJPD $= \sum_{l=1}^{k} \left\{ \max_{J_i, J_j \in G_l} \Delta_{i,j} \right\}$ with $\Delta_{i,j} = \left| [\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)] - [\alpha \cdot \max(t_j^m, t_j^s) + (1-\alpha) \cdot (t_j^m + t_j^s)] \right|$. Here, $\Delta_{i,j}$ denotes the job priority difference between J_i and J_j. The optimal grouping result can be obtained by a dynamic programming approach. Without loss of generality, we assume that all jobs are already sorted according to their priorities, i.e., $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$ is non-decreasing with respect to the index i. Let $OPT_{j,l}$ denote the optimal SMJPD for the first j jobs $(J_1, J_2, ..., J_j)$, when they are divided to l groups. $OPT_{n,k}$ is the desired result. The optimal substructure for the dynamic programming approach is shown as follows:

$$OPT_{j,l} = \min_{l \le i \le j}\{OPT_{i-1,l-1} + \Delta_{i,j}\} \quad (1)$$

Since jobs are assumed to be sorted by their priorities, $\Delta_{i,j}$ is also the maximum job priority difference for the job group of $J_i, J_{i+1}, ..., J_j$. Then, Eq. 1 can be interpreted as follows. The optimal grouping for the first j jobs of l groups is composed of (1) the optimal grouping for the first $i-1$ jobs of $l-1$ groups, and (2) the remaining jobs of $J_i, J_{i+1}, ..., J_j$ as a new group. The index of i is traversed to guarantee the optimality. Since i is traversed, computing the dynamic programming entry of $OPT_{j,l}$ takes $O(n)$ on average. $O(nk)$ entries exist in total, and thus, the eventual time complexity of the dynamic programming approach is $O(n^2k)$. As for the initialization, we set $OPT_{j,l} = 0$ when $j = 0$ or $j \le l$.

Algorithm 4 is proposed, setting the job priority of J_i as $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$ (lines 1 to 3). Then, the dynamic programming approach is applied to group jobs based on their priorities. Meanwhile, groups are also sorted according to their priority ranges (lines 4 and 5). For each group, jobs are iteratively paired according to their map and shuffle workload differences (lines 6 and 7). We prioritize jobs with smaller workloads (the first stage) over jobs with better pairs (the second stage), since the former one generally rules the latter one (verified in experiments). The time complexity of Algorithm 4 is $O(n^2k)$, which results from the dynamic programming approach. Although Algorithm 4 has a higher time complexity than Algorithms 1, 2, and 3, it skips the discretization process, which may result in information loss. As a tradeoff, Algorithm 4 controls the job granularity through a more flexible manner via k.

Algorithm 5 Online Group-based Scheduling Policy

Input: The old schedule, S, and a new arriving job, J_i.
Output: A new schedule of the current job execution order.

1: Set J_i's priority as $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$;
2: **if** a random number is smaller than $1/nk^2$ **then**
3: Call Algorithm 4 to completely reschedule all jobs;
4: **return** the new schedule;
5: **else**
6: **for** each job group, G_l, in S **do**
7: Compute $\max_{J_j \in G_l} \Delta_{i,j}$;
8: Add J_i into $G_l = \arg\min_{G_l}\{\max_{J_j \in G_l} \Delta_{i,j}\}$;
9: Reorder jobs in G_l via the same way as Algorithm 4;
10: **return** the updated S as the schedule;

4.5 Extension to Online Scheduling

The proposed online scheduling algorithm includes an initialization process. At the system start time, Algorithm 4 is used to schedule the existing jobs. If the number of existing jobs is less than k, then each job is regarded as a job group. Note that job groups are sorted according to their priority ranges. Upon a new job arrival, Algorithm 5 is called. It includes two sub-methods: method one completely reschedules all jobs (lines 2 and 3) and method two slightly modifies the existing old schedule (lines 5 to 10). Methods one and two are chosen through a random number generator. The random number is uniformly distributed between 0 and 1. Therefore, line 2 indicates that Algorithm 5 has a small probability of $\frac{1}{nk^2}$ to choose method one, and has a large probability of $1 - \frac{1}{nk^2}$ to choose method two. Here, n is the total number of jobs that are waiting for the schedule. The above probabilities aim to balance the time complexity.

Method one calls Algorithm 4 to reschedule all jobs, and thus takes a time complexity of $O(n^2k)$. In contrast, method two modifies the existing old schedule to resolve the new job. It checks every job group for the new arrival job, and then adds the new job to its closest existing job group. The closest group is the one that can minimize the maximum job priority difference with the new job (lines 6 to 8). It can be found within a time complexity of $O(k)$, since we only need to check the minimum and maximum job priorities in each job group. All jobs in this group and the new arrival job are completely reordered in a pairwise manner (line 9). Since each job group is expected to include $\frac{n}{k}$ jobs, method two is also expected to take $O(\frac{n}{k})$. Consequently, Algorithm 5 takes $O(\frac{n}{k})$, since $[\frac{1}{nk^2} \cdot O(n^2k) + (1 - \frac{1}{nk^2}) \cdot O(\frac{n}{k})] \in O(\frac{n}{k})$.

4.6 Job Prediction and Error Handling

The performance of our algorithms depend on the parameters α, n, and k. However, jobs are no longer known a

priori for the online scenario. At this time, job prediction approaches can be applied to determine these parameters. For example, CherryPick [26] leverages Bayesian optimization to build performance models for various applications, and the models are just accurate enough to distinguish the best or close-to-the-best configuration from the rest with only a few test runs. We can use the same approach to determine the parameters α, n, and k. In addition, the history of job workloads can be used to improve the prediction accuracy. For example, Cortez [27] introduced an extensive characterization of job workloads, including distributions of the job lifetime, deployment size, and resource consumption. Moreover, job prediction can be used to handle errors with respect to job workload information. If the job workload information is not available, we can use predicted job workload to schedule it.

5 Offline Scheduling with Weak Job Pair

Based on the strong job pair, the previous section describes several scheduling algorithms. However, the requirement of the strong job pair is very strict, and thus, is not likely to be satisfied in real scenarios. To relax such a strict requirement, this section further proposes the concept of the weak job pair and the corresponding scheduling algorithms.

5.1 Weak Job Pair and Its Optimality

We start with relaxing the strong job pair to the weak job pair, as defined in the following:

Definition 4. Two jobs, J_i and J_j, are called a weak pair if $t_i^m + t_j^m = t_i^s + t_j^s$.

Definition 5. A weak pair is further called a strict weak pair if the shuffle-heavy job in this weak pair occupies no more than half of the total map workload and no less than half of the total shuffle workload.

The weak pair is an extension of the strong pair. A strong pair must be a weak pair, but vice not versa. The key idea is that the weak pair also avoids I/O underutilization, leading to a better schedule. However, weak pairs do not guarantee the optimality, since pairing a small job and a large job (in terms of total workloads) may not be effective. Such a pairing avoids I/O underutilization at the cost of scheduling a heavier job ahead, as shown in the following example:

Jobs	J_1	J_2	J_3	J_4
t_i^m	1	98	45	55
t_i^s	2	97	49	51

Let us consider two schedules for the above example. The first schedule is just J_1, J_2, J_3, and J_4. Note that J_1 and J_2 form a weak pair, while J_3 and J_4 form another weak pair. Moreover, the weak pair formed by J_3 and J_4 has a larger total workload than the weak pair formed by J_1 and J_2. The first schedule completely avoids I/O underutilization. The job makespan for J_1, J_2, J_3, and J_4 is 2, 99, 148, and 199, respectively. Consequently, the average job makespan is 112. In contrast, the second schedule is J_1, J_3, J_4, and J_2. Note that J_1 and J_3 do not form a weak pair. The job makespan for J_1, J_2, J_3, and J_4 is 2, 51, 102, and 199, respectively. Consequently, the average job makespan is 88.5, which is smaller than 112. Clearly, the first schedule is not optimal. This is simply because the first schedule avoids I/O underutilization at the cost of scheduling a heavier job ahead (i.e., J_2 is scheduled ahead). On the other hand, if weak pairs are formed without scheduling a heavier job ahead, then the optimality can be obtained:

Theorem 4. If J can be decomposed to strict weak pairs of jobs, then jobs that can form a strict weak pair are pairwisely executed in the optimal offline schedule for J. For each strict weak job pair, the shuffle-heavy job is executed before the map-heavy job.

The proof of Theorem 4 is omitted since it is a very simple extension of Theorem 1. The key insight of Theorem 4 is that strict weak pairs are formed without scheduling a heavier job ahead. For each strict weak pair, the total workload of the shuffle-heavy job is close to the total workload of the map-heavy job. The shuffle-heavy job and the map-heavy job in a strict weak pair occupy no less than half of the total shuffle and map workloads, respectively. Compared to the strong pair, the weak pair is more practical. The next two subsections leverage weak pairs for the scheduling.

5.2 New Couple-based Scheduling

Algorithm 6 is proposed to leverage weak pairs of jobs. It is a variation of Algorithm 2 and also needs the discretization process. The idea is to maintain weak pairs in the scheduling. Lines 1 and 2 show the initialization. Line 3 sorts jobs that have the same $|t_i^m - t_i^s|$, i.e., jobs that can form a weak pair are grouped. Note that each job group may have more than two jobs, i.e., a job may form different weak pairs with the other jobs. Lines 4 and 5 process each subset of jobs. The smallest possible pair is iteratively taken out. If a subset of jobs includes an odd number of jobs, then the last job will be put to the end of the schedule. Lines 6 and 7 determine the scheduling based on pairs, which are ordered by their total workloads, $t_i^m + t_j^m + t_i^s + t_j^s$. Finally, line 8 returns the result. Let n denote the number of jobs in J. The time complexity of Algorithm 6 becomes $O(n^2)$. This is because we need $O(n^2)$ to find out the smallest pair in each subset of jobs. Sorting takes only $O(n \log n)$.

Jobs	J_1	J_2	J_3	J_4	J_5	J_6
Discrete t_i^m	3Δ	7Δ	2Δ	3Δ	7Δ	8Δ
Discrete t_i^s	4Δ	6Δ	4Δ	1Δ	5Δ	10Δ

The above table shows an example for Algorithm 6 after the discretization process. In line 3, J_1 and J_2 are sorted together, while J_3, J_4, J_5, and J_6 are sorted together. Note that J_3 can form two different weak pairs with J_4 and J_6, respectively. Lines 4 and 5 form weak job pairs for each subset of jobs. In the first subset, J_1 and J_2 are paired. In the second subset, J_3 and J_4 are paired in the first iteration, since they form the smallest weak pair (compared to J_3 and J_5, J_6 and J_4, and J_6 and J_5). J_6 and J_5 are paired in the second iteration (shuffle-heavy job before map-heavy job for each weak pair). Line 5 sorts these three weak pairs into J_3 and J_4, J_1 and J_2, and J_6 and J_5. Consequently, the final schedule is J_3, J_4, J_1, J_2, J_6, and J_5.

Algorithm 6 is a variation of Algorithm 2. The former is based on weak pairs while the latter is based on strong

Algorithm 6 New Couple-based Scheduling Policy

Input: The job set, J, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.

1: Initialize an array, S, to denote the job execution order;
2: Put all jobs into the order array of S;
3: Sort all jobs in S according to $|t_i^m - t_i^s|$;
4: **for** each subset of jobs with the same $|t_i^m - t_i^s|$ **do**
5: Pair jobs by iteratively taking out a pair, J_i and J_j, such that $t_i^m + t_i^s + t_j^m + t_j^s$ is the smallest among all possible pairs in the job subset;
6: Sort all pairs by their total workloads, $t_i^m + t_i^s + t_j^m + t_j^s$;
7: Put sorted pairs into S; in each pair, the job that maximizes $t_i^s - t_i^m$ is scheduled first;
8: **return** the order array of S as the schedule;

Algorithm 7 Match-based Scheduling Policy

Input: The job set, J, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.

1: Initialize an array, S, to denote the job execution order;
2: **if** the number of jobs in J is odd **then**
3: Find out the most map-heavy job that maximizes $t_i^m - t_i^s$, put it to the end of S, and remove it from J.
4: **for** each pair of jobs in J, say J_i and J_j **do**
5: Define the matching weight between jobs J_i and J_j as $w_{ij} = \alpha \cdot |t_i^m + t_j^m - t_i^s - t_j^s| + (1-\alpha) \cdot |t_i^m + t_i^s - t_j^m - t_j^s|$.
6: Find out the minimum weighted matching for jobs in J through the weighted Blossom algorithm; Two matched jobs are regarded as a matched pair;
7: Sort all pairs by their total workloads, $t_i^m + t_i^s + t_j^m + t_j^s$;
8: Put sorted matched pairs into S; in each pair, the job that maximizes $t_i^s - t_i^m$ is scheduled first;
9: **return** the order array of S as the schedule;

pairs. Note that Algorithm 6 is not necessarily better than Algorithm 2 in terms of the average job makespan. This is because weak pairs may be formed at the cost of scheduling a heavier job ahead. Actually, all proposed algorithms are trying to balance the tradeoff between the job pairing and the job workload. The next subsection uses matching to control the above tradeoff without the discretization process.

5.3 Match-based Scheduling Policy

This subsection uses the matching to balance the tradeoff between the job pairing and the job workload without the discretization process. The key idea is that, when two jobs are paired, we need to consider both their pairing degree and their workload difference. The pairing degree measures to what degree two jobs can form a weak pair, such that I/O underutilization can be avoided. On the other hand, a smaller workload difference between two jobs gets rid of scheduling a heavier job ahead. We use the matching weight to check whether two jobs can be paired. Given two jobs, J_i and J_j, their matching weight is defined as follows:

$$w_{ij} = \alpha \cdot |t_i^m + t_j^m - t_i^s - t_j^s| + (1-\alpha) \cdot |t_i^m + t_i^s - t_j^m - t_j^s| \quad (2)$$

The former part, $|t_i^m + t_j^m - t_i^s - t_j^s|$, is the pairing degree. The latter part, $|t_i^m + t_i^s - t_j^m - t_j^s|$, is the workload difference between J_i and J_j. α is just a weight parameter that satisfies

Algorithm 8 Online Match-based Scheduling Policy

Input: The old schedule, S, and a new arriving job, J_i.
Output: A new schedule of the current job execution order.

1: **if** a random number is smaller than $1/n^4$ **then**
2: Call Algorithm 7 to completely reschedule all jobs;
3: **return** the new schedule;
4: **else**
5: **for** each job in J, say J_j **do**
6: Define the matching weight between J_i and J_j as $w_{ij} = \alpha \cdot |t_i^m + t_j^m - t_i^s - t_j^s| + (1-\alpha) \cdot |t_i^m + t_i^s - t_j^m - t_j^s|$.
7: Find the job, say J_j, that minimizes w_{ij};
8: **if** $t_i^s - t_i^m > t_j^s - t_j^m$ **then**
9: Put J_i before J_j and update S.
10: **else**
11: Put J_i after J_j and update S.
12: **return** the updated S as the schedule;

$0 \leq \alpha \leq 1$ (also used in Algorithms 3, 4, and 5). Note that a smaller w_{ij} indicates a better pairing between J_i and J_j. w_{ij} reduces to 0 when J_i and J_j form a strong pair. When $\alpha = 1$, the matching weight only considers the job pairing degree. In this case, jobs that can form weak pairs have the minimum matching weight. On the other hand, when $\alpha = 0$, the matching weight only considers the job workload difference. In this case, jobs with the same workloads have the minimum matching weight.

The minimum weighted matching, which can be computed through the weighted Blossom algorithm [28], is used to balance the tradeoff between the job pairing degree and the job workload difference. Consequently, Algorithm 7 is proposed. Line 1 is the initialization. Lines 2 and 3 focus on a corner case, in which J includes an odd number of jobs. In this corner case, the most map-heavy job, which maximizes $t_i^m - t_i^s$, is scheduled to the end of S. Lines 4 and 5 compute the matching weight for each pair of jobs in J. Consequently, line 6 pairs jobs through the minimum weighted matching. Lines 7 and 8 sort and schedule matched job pairs by their total workloads, $t_i^m + t_i^s + t_j^m + t_j^s$. In each pair, the job that maximizes $t_i^s - t_i^m$ is scheduled first. Finally, line 9 returns the result. The time complexity of Algorithm 7 is $O(n^4)$ due to the weighted Blossom algorithm [28–30].

5.4 Extension to Online Scheduling

This subsection extends the match-based scheduling policy to the online scenario, where jobs are no longer known a priori. The scheduler can only obtain the workload information of a job upon its arrival. Due to the problem hardness, we propose a heuristic scheduling algorithm based on Algorithm 7. The key idea is similar to Algorithm 5, which uses a probabilistic procedure to balance the tradeoff between the algorithm performance and the time complexity.

The proposed online scheduling algorithm includes an initialization process. At the system start time, Algorithm 7 is used to schedule the existing jobs. Upon a new job arrival (say J_i), Algorithm 8 is called. It includes two submethods: method one completely reschedules all jobs (lines 1 to 3), and method two just inserts J_i into the existing old schedule (lines 4 to 11). Methods one and two are chosen through a random number generator. The random number

is uniformly distributed between 0 and 1. Therefore, line 1 indicates that Algorithm 8 has a small probability of $\frac{1}{n^4}$ to choose method one, and has a large probability of $1 - \frac{1}{n^4}$ to choose method two. Here, n is the total number of jobs that are waiting for the schedule.

Method one calls Algorithm 7 to reschedule all jobs, and thus takes a time complexity of $O(n^4)$. In contrast, method two modifies the existing old schedule to resolve the new job. It computes the matching weight between the newly arrived job, J_i, and each existing job in J (lines 5 and 6). The job, say J_j, that has the minimum matching weight with J_i is found in line 7. J_i will be inserted around J_j, depending on their workloads. If J_i is more shuffle-heavy (i.e., $t_i^s - t_i^m > t_j^s - t_j^m$), J_i is inserted before J_j in S (lines 8 and 9). On the other hand, if J_j is more shuffle-heavy (i.e., $t_i^s - t_i^m \leq t_j^s - t_j^m$), J_i is inserted after J_j in S (lines 10 and 11). Since lines 5 and 6 need to compute the matching weight between the newly arrived job and each existing job, method two takes $O(n)$. Consequently, Algorithm 8 takes $O(n)$, since $[\frac{1}{n^4} \cdot O(n^4) + (1 - \frac{1}{n^4}) \cdot O(n)] \in O(n)$. Note that method one has a better scheduling performance at the cost of a larger time complexity, while method two has a worse scheduling performance but a smaller time complexity. They are balanced through the random number generator, leading to a linear time complexity on expectation.

6 Experiments

6.1 Settings

Our experiments are conducted based on the Google cluster dataset [31, 32], which are described in [25]. Four algorithms are used for comparison:

- MaxDiff ranks jobs by their map and shuffle workload differences ($t_i^m - t_i^s$ for job J_i). The job with a larger workload difference will be executed later. It prioritizes shuffle-heavy jobs over map-heavy jobs to avoid I/O resource underutilization.
- Pairwise is based on Theorem 1, which has suggested that jobs should be pairwisely scheduled. This policy orders jobs by iteratively taking out a pair of jobs of $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$.
- MaxShuffle ranks jobs by their shuffle workloads. Jobs with a larger shuffle workload are executed earlier in order to avoid I/O resource underutilization.
- MaxSRPT is proposed by Lin et al. [16]. It schedules jobs according to their dominant workloads, i.e., $\max(t_i^m, t_i^s)$ for job J_i. Our algorithms improve MaxSRPT through executing jobs pairwisely.

Our experiments denote Algorithms 1 to 8 as Pair-based, Couple-based, Generalized, Group-based, OGroup-based, NCouple-based, Match-based, and OMatch-based scheduling policies for simplicity. In default, we can set $\Delta = 0.1$ seconds as the discretization step (Algorithms 1, 2, 3, and 6), $\alpha = 0.5$ as the weight parameter (Algorithms 3, 4, 5, 7, and 8), and $k = 20$ as the number of groups (Algorithms 4 and 5). Three metrics are used for comparison. The first metric is the average job makespan, which is the time span from the job arrival to its shuffle phase completion. The other two metrics are the *average job waiting time* and the *average job execution time*. The waiting time of a job is the time span from the job arrival to the start of its map phase. The execution time of a job is the time span from the start of its map phase to the completion of its shuffle phase. The job makespan is the sum of the job waiting time and the job execution time.

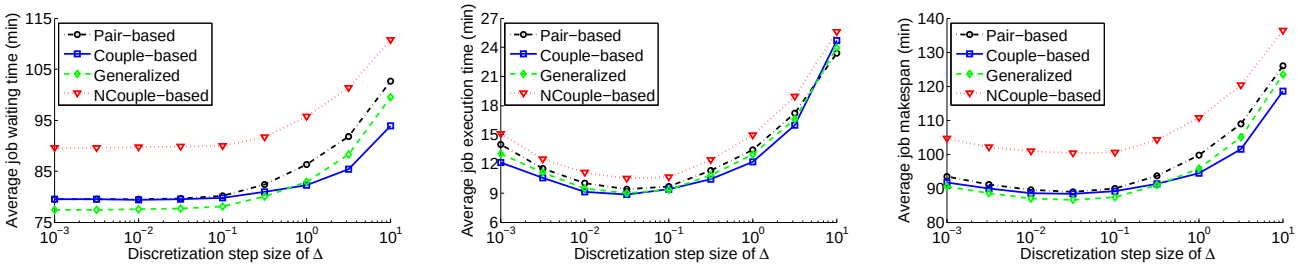
TABLE 2
Offline performance evaluation in the Google cluster dataset.

Scheduling algorithms	Average job waiting time	Average job execution time	Average job makespan
MaxDiff	8806	682	9488
Pairwise	8289	149	9138
MaxShuffle	7929	898	8827
MaxSRPT	4768	840	5608
Pair-based	4809	581	5390
Couple-based	4787	563	5350
Generalized	4683	560	5243
Group-based	4619	532	5151
NCouple-based	5399	636	6035
Match-based	4431	512	4943

6.2 Evaluation Results for Offline Scheduling

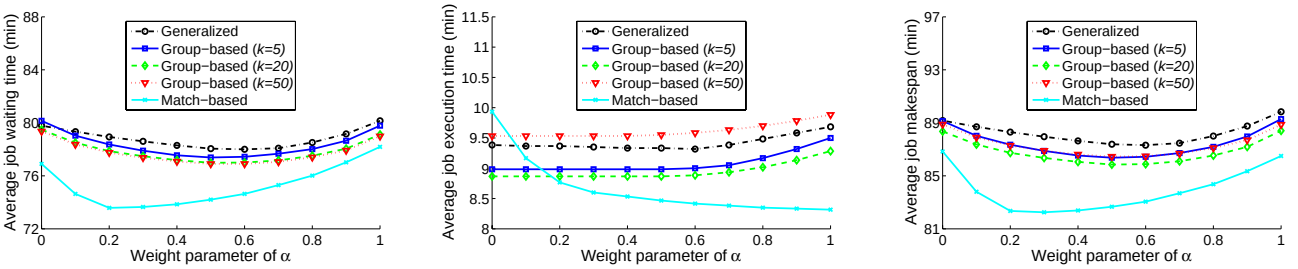
Experiments in the Google cluster dataset are conducted for the offline scenario, in which all jobs are supposed to arrive at the system start time. The results are shown in Table 2 with the unit of seconds. MaxDiff, Pairwise, and MaxShuffle have the worst performances. However, Pairwise has a significant smallest average job execution time through executing jobs pairwisely. It ignores the total map and shuffle workloads of jobs, leading to an overly large job waiting time. We also find that the Pair-based scheduling policy has a larger average job wait time than the MaxSRPT policy, since the discretization process is information-lossy. However, the former policy has a smaller average job execution time through executing jobs pairwisely. The Couple-based policy improves the Pair-based policy through considering the total map and shuffle workloads of a job rather than its dominant workload. The Generalized policy improves the Pair-based and the Couple-based policies by combining them with a given weight parameter of α. The Group-based policy improves the Generalized policy by grouping jobs optimally. An interesting observation is that the NCouple-based policy does not have a good performance. This is because weak pairs are formed at the cost of scheduling a heavier job ahead. For the Google cluster dataset, we can conclude that scheduling jobs with lighter workloads before jobs with heavier workloads is more important than avoiding I/O underutilization through weak pairs. Finally, the Match-based policy has the best performance, since it subtly balances the tradeoff between the job pairing and the job workload through a matching procedure.

The impacts of the discretization step size, Δ, is shown in Fig. 2 (offline scenario in the Google cluster dataset). Fig. 2(a) shows that a small Δ does not have a significant impact on the average job waiting time. However, a large Δ results in an exponentially increased average job waiting time, due to the information loss on the total or dominant job workload. Meanwhile, Fig. 2(b) shows that both overly small and overly large Δ will increase the job execution time. This is because the pairing process is broken down by an improper Δ. The corresponding average job makespan is shown in Fig. 2(c). We can conclude that Δ should not be overly small or overly large to minimize the average job makespan. For the Google cluster dataset, a good value for Δ can range from 0.01 to 0.1.



(a) Average job waiting time. (b) Average job execution time. (c) Average job makespan.

Fig. 2. Offline performance evaluation with respect to the discretization step size of Δ.



(a) Average job waiting time. (b) Average job execution time. (c) Average job makespan.

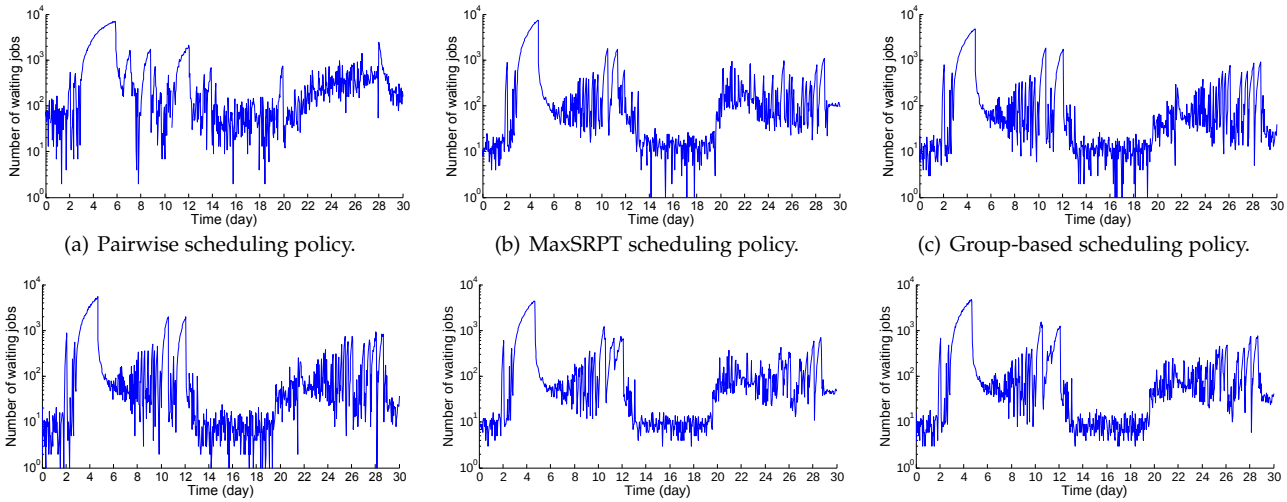
Fig. 3. Offline performance evaluation with respect to the weight parameter α.

The impacts of the weight parameter, α, and the group number, k, are together shown in Figs. 3 (offline scenario in the Google cluster dataset). As for the weight parameter α, Fig. 3(a) shows an interesting pattern. The Generalized policy reduces to the Pair-based policy when $\alpha = 1$, and reduces to the Couple-based policy when $\alpha = 0$. However, it achieves the smallest average job waiting time when α is around 0.6. In addition, the Match-based policy is significantly different than all the other policies with respect to α, and it achieves the smallest average job waiting time when α is around 0.2. The average job execution time is shown in Fig. 3(b). α has a slight impact on the average job execution time for the Generalized and Group-based policies, but has a significant impact for Match-based policy. This is because, when $\alpha = 1$ in the Match-based policy, the matching weight only considers the job pairing degree, such that I/O underutilization can be avoided at the cost of scheduling a heavier job ahead. The average job makespan is shown in Fig. 3(c). The Match-based policy has a smaller average job makespan than the Generalized and Group-based policies. Another notable point is with respect to k. While Fig. 3(a) shows that an overly small k leads to a large average job wait time, and Fig. 3(b) shows that an overly large k leads to a large average job execution time. As shown in Fig. 3(c), in order to minimize the average job makespan for the Group-based policy, k should be neither too small nor too large, depending on the dataset.

6.3 Evaluation Results for Online Scheduling

Experiments in the online scenario are conducted in the Google cluster dataset, which includes the job arrival time. We start with the number of waiting jobs per hour under each scheduling policy. The results are shown in Fig. 4. Not all policies are presented here due to the page limitation. Fig. 4(a) shows the result for the Pairwise policy, which has the worst performance. Compared to other policies, the Pairwise policy has a larger number of waiting jobs for a longer time around days 4, 5, and 12. It also has more waiting jobs from days 22 to 30. Fig. 4(b) shows the result for the MaxSRPT policy, which is not the best one, due to the peak for days 20 to 24. In contrast to the Pairwise and MaxSRPT policies, the Group-based policy has a smaller number of waiting jobs over time, as shown in Fig. 4(c). This is because it considers to schedule jobs in a pairwise manner to avoid the underutilization of the I/O resource. The performance of the OGroup-based policy is shown in Fig. 4(d). It has a slightly worse performance from days 26 to 30 than its offline version. Note that the scheduling time complexity of the online version is $O(\frac{n}{k})$, which is lower than the scheduling time complexity of the offline version, $O(n^2k)$. The performance of the Match-based policy is shown in Fig. 4(e). The Match-based policy has the best performance among all policies, since it balances the tradeoff between the job pairing and the job workload. However, the time complexity for the Match-based policy is $O(n^4)$ upon each new job arrival, in which n is the number of waiting jobs. To reduce the scheduling complexity, the OMatch-based policy is introduced, as shown in Fig. 4(f). The OMatch-based policy has a slightly worse performance than Match-based policy. However, the time complexity of the OMatch-based policy is $O(n)$, which is significantly smaller than the time complexity of the Match-based policy.

Detailed performance statistics are shown in Table 3 with the unit of minutes. A new concept of an 80%-interval is used to represent the variable interval after removing the largest 10% of the values and the smallest 10% of the values. For each job, we define its ratio as the ratio of its practical execution time to its dominant workload. A larger ratio indicates that the corresponding job needs to wait for the I/O resource for a relatively longer time. A job with a ratio of one means that its execution time cannot be shortened. The



(a) Pairwise scheduling policy. (b) MaxSRPT scheduling policy. (c) Group-based scheduling policy.

(d) OGroup-based scheduling policy. (e) Match-based scheduling policy. (f) OMatch-based scheduling policy.

Fig. 4. Online performance evaluation with respect to the number of waiting jobs.

TABLE 3
Online performance evaluation in the Google cluster dataset.

Scheduling algorithms	Job waiting time statistics			Job execution time statistics			Job makespan statistics			
	Average	80%-interval	Max	Average	80%-interval	Worst ratio	Average	80%-interval	Min	Max
MaxDiff	42	[1, 138]	220	5	[0, 13]	239	47	[0, 138]	0	230
Pairwise	35	[1, 103]	193	1	[0, 3]	45	36	[0, 101]	0	178
MaxShuffle	23	[0, 86]	176	15	[2, 37]	542	38	[3, 95]	1	154
MaxSRPT	16	[0, 45]	183	13	[0, 44]	220	28	[0, 74]	0	81
Pair-based	19	[1, 51]	205	3	[0, 5]	113	22	[1, 48]	1	54
Couple-based	18	[1, 49]	195	3	[0, 5]	75	21	[1, 46]	1	49
Generalized	16	[1, 44]	187	3	[0, 5]	69	19	[1, 45]	1	47
Group-based	15	[1, 41]	174	3	[0, 4]	58	18	[1, 39]	1	41
OGroup-based	16	[2, 47]	185	3	[0, 5]	68	19	[1, 46]	1	48
NCouple-based	21	[2, 63]	201	1	[0, 3]	51	32	[1, 56]	0	97
Match-based	12	[1, 38]	164	3	[0, 4]	54	15	[1, 37]	1	39
OMatch-based	13	[2, 40]	178	3	[0, 4]	61	16	[1, 42]	1	44

worst ratio is the smallest ratio among all jobs. In Table 3, it can be seen that the Pairwise policy has the smallest job execution time, but it has a larger job waiting time. In contrast, the MaxSRPT policy has a small job waiting time with a large job execution time. The proposed Pair-based and Couple-based policies balance the job waiting and execution times to obtain smaller job makespans. The Generalized policy has a slight improvement by combining the above two policies, while the Group-based policy also has some improvements by optimally grouping jobs. The OGroup-based policy has a slightly worse performance than Group-based policy, as well as a smaller scheduling time complexity. Note that the NCouple-based policy does not have a good performance. Although its job execution time is small, its job waiting time is large. The Match-based policy has the best performance by balancing the tradeoff between the job pairing and the job workload. Similar to the OGroup-based policy, the OMatch-based policy also has a slightly worse performance than the Match-based policy, as well as a smaller scheduling time complexity.

7 Acknowledgments

This research was supported in part by NSF grants CNS 1629746, CNS 1564128, CNS 1449860, CNS 1461932, CNS 1460971, and CNS 1439672.

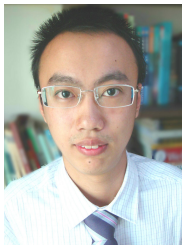
8 Conclusion

MapReduce includes three phases of map, shuffle, and reduce. Since the map phase is CPU-intensive and the shuffle phase is I/O-intensive, these phases can be conducted in parallel. This paper focuses on a joint scheduling optimization in MapReduce, where map and shuffle phases can be overlapped and be conducted in parallel. The scheduling objective is to minimize the average job makespan. The key challenge is that the map and shuffle phases cannot be fully parallelized due to their dependency relationship: the shuffle phase may wait to transfer the data emitted by the map phase. To avoid I/O underutilization, jobs that can form a strong pair should be pairwisely executed. Several offline and online scheduling policies are proposed to execute jobs in a pairwise manner. Scheduling optimalities are discussed under several scenarios. We also explore scheduling policies based on weak pairs, in terms of balancing the tradeoff between the job pairing and the job workload. Finally, real data-driven experiments validate the efficiency and effectiveness of the proposed scheduling policies.

References

[1] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2011, pp. 390–399.

[2] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 29–42.
[3] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu, "Mimp: Deadline and interference aware scheduling of hadoop virtual machines," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid (CCGrid)*, 2014, pp. 394–403.
[4] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *European Conference on Computer Systems (EuroSys)*, 2010, pp. 265–278.
[5] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2011, pp. 235–244.
[6] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo, "Automated profiling and resource management of pig programs for meeting service level objectives," in *IEEE International Conference on Autonomic Computing (ICAC)*, 2012, pp. 53–62.
[7] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, "Flex: A slot allocation scheduling optimizer for mapreduce workloads," in *ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2010, pp. 1–20.
[8] Z. Tang, L. Jiang, J. Zhou, K. Li, and K. Li, "A self-adaptive scheduling algorithm for reduce start time," *Future Generation Computer Systems*, vol. 43, pp. 51–60, 2015.
[9] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in mapreduce clusters using mantri." in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 24–33.
[10] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: optimizing mapreduce on heterogeneous clusters," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 61–74, 2012.
[11] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 261–276.
[12] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, I. Stoica, I. Y. L. Dont, and B. Us, "True elasticity in multi-tenant clusters through amoeba," in *ACM Symposium on Cloud Computing (SoCC)*, 2012, pp. 1–7.
[13] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014, pp. 455–466.
[14] Q. Zhang, M. F. Zhani, Y. Yang, R. Boutaba, and B. Wong, "Prism: Fine-grained resource-aware scheduling for mapreduce," *IEEE Transactions on Cloud Computing*, vol. 3, no. 2, pp. 182–194, 2015.
[15] A. Verma, B. Cho, N. Zea, I. Gupta, and R. H. Campbell, "Breaking the mapreduce stage barrier," *Journal of Cluster Computing*, vol. 16, no. 1, pp. 191–206, 2013.
[16] M. Lin, L. Zhang, A. Wierman, and J. Tan, "Joint optimization of overlapping phases in mapreduce," *Performance Evaluation*, vol. 70, no. 10, pp. 720–735, 2013.
[17] J. Li, J. Wu, and X. Yang, "Optimizing mapreduce based on locality of kv pairs and overlap between shuffle and local reduce," in *International Conference on Parallel Processing (ICPP)*, 2015, pp. 1–10.
[18] J. Wang, M. Qiu, B. Guo, and Z. Zong, "Phase reconfigurable shuffle optimization for hadoop mapreduce," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2015.
[19] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "Omo: Optimize mapreduce overlap with a good start (reduce) and a good finish (map)," in *International Performance Computing and Communications Conference (IPCCC)*, 2015, pp. 1–8.
[20] Q.-K. Pan, M. F. Tasgetiren, P. N. Suganthan, and T. J. Chua, "A discrete artificial bee colony algorithm for the lot-streaming flow shop scheduling problem," *Information Sciences*, vol. 181, no. 12, pp. 2455–2468, 2011.
[21] N. Lim, S. Majumdar, and P. Ashwood-Smith, "Mrcp-rm: A technique for resource allocation and scheduling of mapreduce jobs with deadlines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1375–1389, 2017.
[22] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "G: Packing and dependency-aware scheduling for data-parallel clusters," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 81–91.
[23] S.-y. Wang, L. Wang, M. Liu, and Y. Xu, "An effective estimation of distribution algorithm for solving the distributed permutation flow-shop scheduling problem," *International Journal of Production Economics*, vol. 145, no. 1, pp. 387–396, 2013.
[24] M. Marichelvam, T. Prabaharan, and X.-S. Yang, "Improved cuckoo search algorithm for hybrid flow shop scheduling problems to minimize makespan," *Applied Soft Computing*, vol. 19, pp. 93–101, 2014.
[25] H. Zheng, Z. Wan, and J. Wu, "Optimizing mapreduce framework through joint scheduling of overlapping phases," in *International Conference on Computer Communication and Networks (ICCCN)*, 2016, pp. 1–9.
[26] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics." in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 469–482.
[27] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 153–167.
[28] R. Duan, S. Pettie, and H.-H. Su, "Scaling algorithms for weighted matching in general graphs," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2017, pp. 781–800.
[29] Z. Guan, G. Si, X. Zhang, L. Wu, N. Guizani, X. Du, and Y. Ma, "Privacy-preserving and efficient aggregation based on blockchain for power grid communications in smart communities," *IEEE Communications Magazine*, vol. 56, no. 7, pp. 1–7, 2017.
[30] Z. Guan, J. Li, L. Wu, Y. Zhang, J. Wu, and X. Du, "Achieving efficient and secure data acquisition for cloud-supported internet of things in smart grid," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1934–1944, 2017.
[31] J. Wilkes, "More Google cluster data," Google research blog, Nov. 2011, posted at http://googleresearch.blogspot.com/2011/11/.
[32] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," white paper, Google Inc., Nov. 2011, posted at http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.



Huanyang Zheng received his B.Eng. degree in Telecommunication Engineering from Beijing University of Posts and Telecommunications, China in 2012. He is currently a Ph.D. candidate in the Department of Computer and Information Sciences, Temple University, USA. His research focuses on wireless and mobile networks, social networks and structures, and cloud systems.

Jie Wu is the Associate Vice Provost for International Affairs at Temple University. He also serves as Director of Center for Networked Computing and Laura H. Carnell professor in the Department of Computer and Information Sciences. Prior to joining Temple University, he was a program director at the National Science Foundation and was a distinguished professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Service Computing and the Journal of Parallel and Distributed Computing. Dr. Wu was general cochair/chair for IEEE MASS 2006, IEEE IPDPS 2008, IEEE ICDCS 2013, and ACM MobiHoc 2014, as well as program co-chair for IEEE INFOCOM 2011 and CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.