Scheduling Jobs Across Geo-distributed Datacenters

Chien-Chun Hung, Leana Golubchik, Minlan Yu
University of Southern California
{chienchun.hung, leana, minlanyu}@usc.edu

Abstract

With growing data volumes generated and stored across geo-distributed datacenters, it is becoming increasingly inefficient to aggregate all data required for computation at a single datacenter. Instead, a more recent trend is to distribute computation to meet data locality, thus reducing the resource (e.g., bandwidth) costs while improving performance. Consequently, new challenges are emerging in job scheduling, where each job runs across multiple geo-distributed datacenters and, requiring coordination among datacenters. In this paper, we propose novel job scheduling algorithms that coordinate job scheduling across datacenters with low overhead, while achieving near-optimal performance. Our extensive simulation study, using realistic job traces, shows that the proposed scheduling algorithms result in up to 50% improvement in average job completion time over the Shortest Remaining Processing Time (SRPT) based approaches.

1. Introduction

Data intensive jobs run by cluster computing systems (e.g., Hadoop[3], Spark[39], Dryad[17]) have recently generated significant workloads for datacenters, providing services such as web search, consumer advertisements and product recommendations, user behavior analysis and business intelligence. These jobs are composed of numerous tasks. Each task reads a partition of input data and runs on available computing slots in parallel; the job is finished upon the completion of all of its tasks [5, 7, 8]. To serve the increasing demands of various data analytics applications, major cloud providers like Amazon[1], Microsoft[4] and Google[19] each deploy from tens to hundreds of geo-distributed datacenters; AT&T has thousands of datacenters at their PoP locations.

Conventional approaches perform *centralized job execution*, with each job running within a single datacenter. In such case, when a job needs data from *multiple* datacenters, a typical approach is to first collect all the required data from multiple datacenters at a single location, and then run the computation at that datacenter [11, 13, 16, 20, 25]. However, as data volumes continue to grow in an unprecedented manner, such an approach results in the substantial network traffic [27, 35, 36] and the increased job completion time [14]. Moreover, it is becoming increasingly impractical to replicate a large data set across multiple datacenters [24]. Finally, some data are restricted to certain datacenters due to security and privacy constraints (e.g., must be kept within a particular nation [35, 36]), and therefore cannot be moved.

Consequently, instead of data aggregation at a single data center, a recent trend is to conduct *distributed job execution*, i.e., running a job's tasks at the datacenters where the needed data are stored, and only aggregating the results at job completion time. Recent research efforts show that distributed job execution achieves $250\times$ bandwidth savings [35, 36] and reduces the $90th-percentile$ job completion time by a factor of 3 [14]; moreover, $3-19\times$ query speed-up and $15-64\%$ reduction in bandwidth costs can be achieved [26].

Although promising, distributed job execution poses new challenges for job scheduling. Since a job's completion time is determined by its last completed task across the datacenters, finishing a portion of the job quickly at one datacenter does not necessarily result in faster overall job completion time. In addition, potential skews in number of tasks per job processed at a particular datacenter (as determined by the data stored there) further complicate matters. Hence, prioritizing a job's tasks at one datacenter when its counterparts at other datacenters dominate the overall job completion is "wasteful" (in the sense that prioritizing a different job may have led to better overall average completion time).

Consequently, unlike in the single-server-single-queue scenario, classical Shortest Remaining Processing Time (SRPT) scheduling [9, 30, 31] fails to optimize the average job completion time in the case of multiple datacenters with parallel task execution. To provide insight into sub-optimal behavior of SRPT (and its natural extensions to the multiple datacenter scenario), we present motivating examples in Section 2, and then show in Section 5 that SRPT-type techniques' scheduling of jobs based only on their sizes results in even worse behavior under heterogeneous datacenters.

To address the challenges outlined above, in this paper, we focus on job scheduling algorithms designed for the multi-datacenter parallel task execution scenario. Even single-server-single-queue versions of this scheduling problem have been shown to be strongly NP-hard [29] or APX-hard [12]. Thus, our efforts are focused on principled heuristic solutions that can be (experimentally) shown to provide near-optimal performance. Specifically, our contributions can be summarized as follows.

- We illustrate why natural SRPT-based extensions leave significant room for performance improvements, which provides insights for better approaches. (Section 2)
- We propose a light-weight "add-on", termed *Reordering*, that can be easily added to any scheduling algorithm to improve its performance by delaying parts of certain jobs without degrading their response times, while providing opportunities for other jobs to finish faster. We prove that executing *Reordering* after any scheduling algorithm would result in performance that is not worse than the one without *Reordering*. (Section 3)
- We construct three principles for designing a job scheduling algorithm aimed at reducing the average job completion time in our setting. Armed with these design principles, we develop *Workload-Aware Greedy Scheduling (SWAG)*, that greedily serves the job that finishes the fastest by taking existing workload at the local queues into consideration. (Section 4)
- As a proof of concept, we implement a prototype using our proposed algorithms under Spark [39] while addressing several system implementation issues (Section 5). We also conduct extensive large-scale simulation-based experiments using realistic job traces under a variety of settings (Section 6). Our results show that *SWAG* and *Reordering* achieve as high as 50% and 27% improvements, respectively, in average job completion time as compared to the SRPT-based extensions. The results also show that the proposed techniques achieve completion times within 2% of an optimal solution (as obtained through brute-force for comparison purposes), while requiring reasonable communication and computation overhead.

2. Background and Motivation

In this section, we first present an overview of the distributed job execution framework in a geo-distributed datacenter system. Next we provide a motivating example to illustrate the needs for better scheduling approaches.

2.1 Job Scheduling across Geo-distributed Datacenters

Figure 1 depicts the general framework for distributed job execution in geo-distributed datacenters. Our system consists of a central controller and a set of datacenters D span-

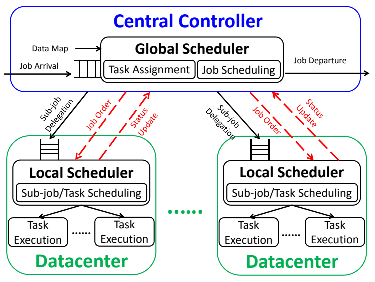


Figure 1. System Architecture of Distributed Job Execution

ning geographical regions, while the system serves the jobs running with input data stored across the geo-distributed datacenters. Each job (arriving at the central controller) is composed of small tasks that process independent input partitions and run in parallel [5, 7, 8].

The main focus of this paper is the development of an effective job scheduling mechanism for geo-distributed datacenters. In our system, job scheduling decisions are made (and potentially re-evaluated) at job arrival and departure instants [1] , and involve two levels of schedulers: (1) The global scheduler residing in the central controller, makes job-level scheduling decisions for all jobs in the system[2], and assigns a job's tasks to the datacenters that host the input data. [3] (2) The local scheduler at each datacenter has a queue q_d that stores the tasks assigned by the global scheduler, and launches the tasks at the next available computing slot based on the job order determined by the global scheduler (or the local scheduler itself). In addition, all datacenters report their progress to the central controller, in support of global job scheduling decisions. The job-level scheduling decisions are therefore made by the coordination of the global and local schedulers (depending on the scheduling technique as described later) and are a function of the set of current jobs J, their tasks, and local queue information data reported by the datacenters. A job is considered completed only after all of its tasks are finished; therefore the job completion time is determined by its last completed task. Our goal is to reduce the average job completion time.

Fully replicating data across all datacenters in today's systems is quite costly, in terms of storage space and in overhead for maintaining consistency among the copies [24]. Instead, recent systems [24] opt for a single primary copy plus multiple partial copies based on coding techniques and replication policies. In our system, each task is assigned to the datacenter that holds its primary copy of the input data. We refer to the subset of the job's tasks assigned to the same datacenter as the job's *sub-job* at that datacenter. Let

[1] We illustrate later in Section 5 that this is sufficient.

[2] In some cases the global scheduler delegates the job-level scheduling to the local schedulers as discussed later.

[3] Some local jobs may go directly to the datacenter where all of their required data is located. We assume that each datacenter reports information about local jobs to the central controller as the jobs arrive.

$v_{j,d}$ denote the sub-job composed of job j's tasks that are assigned to datacenter d. The order in which these sub-jobs are served at each data center is determined by the job-level scheduling decisions, where the local scheduler continues launching the task of the first sub-job in the queue whenever a computing slot becomes available unless the order of sub-jobs is updated. When such modifications occur, we assume no preemption for a task execution when it's running [4] , but a job (or sub-job) execution can be preempted, i.e., the tasks of other jobs (or sub-jobs) can be scheduled to run before the non-running tasks of the currently running job (or sub-job).

To facilitate global scheduling decisions, each datacenter reports its current snapshot (including the progress of the sub-jobs in service and those in the queue) to the central controller. For simplicity of presentation and evaluation, we assume that this information is guaranteed to be delivered in time and accurate. In addition, we assume that our system primarily serves the jobs with single-stage tasks; we discuss how our system can be extended to serve the jobs with multi-stage tasks in Section 8.

2.2 Motivating Example

We now present a simple example to illustrate how the various scheduling techniques work and the differences of their scheduling results. Table 1 describes the example settings (job arrival order, number of tasks per job and their distribution among the data centers); Figure 2 provides the scheduling results obtained by the various scheduling techniques described in this paper. In this example, there are three jobs arriving to the system at different times, with Job A followed by Job B, followed by Job C. At the time the scheduler makes the scheduling decision, these three jobs all have some tasks that are not yet launched. The jobs' remaining sizes[5] in each datacenter are also given in Table 1. In this example each datacenter has a single compute slot, i.e., the datacenter serves one task at a time.

Let the completion time of job i be $r_i = f_i - a_i$, where f_i and a_i are the time instants of finishing the job i (or, finish time) and job i's arrival, respectively. Then, the average job completion time of n jobs is $\frac{1}{n} \times \sum_{i=1}^{n} r_i = \frac{1}{n} \times \sum_{i=1}^{n} (f_i - a_i) = \frac{1}{n} \times \{\sum_{i=1}^{n} f_i - \sum_{i=1}^{n} a_i\}$. We can view reducing the average job completion time as reducing the sum of the finish times, $\sum_{i=1}^{n} f_i$ (or equivalently, $\frac{1}{n} \times \sum_{i=1}^{n} f_i$), as $\sum_{i=1}^{n} a_i$ is constant. For simplicity of exposition, we discuss the remainder of the example in terms of reducing average finish time (rather than the average completion time).

We further define a sub-job's *finish instant* $i_{j,d}$ as the queue index at which sub-job $v_{j,d}$ ends, which is computed as $i_{z,d} + |v_{j,d}|$, where $v_{z,d}$ is the sub-job that is right next to $v_{j,d}$ while being earlier in the queue, and $|v_{j,d}|$ is the size (remaining number of tasks) of sub-job $v_{j,d}$. The sub-job's finish instant is a relative measure and a monotonic indicator of its finish time; [6] specifically, given that $i_{a,d} < i_{b,d} \forall a, b \in J$, sub-job $v_{a,d}$ finishes no later than sub-job $v_{b,d}$ does. In addition, a job's finish instant is the maximum finish instant of all its sub-jobs, i.e., $\max_d i_{j,d}, \forall d \in D$. In this example, if we were to use a First Come First Serve (FCFS) scheduling approach, the finish instants of Jobs A, B, and C would be 10, 18 and 11, respectively, which results in an average job finish instant of 13.

[4] Non-preemptive task execution is common in conventional cluster computing systems [3, 39] as the tasks are typically of short duration and hence switching cost is (relatively) large.

[5] Here, a job's remaining size is its remaining number of tasks that are not launched yet.

2.3 SRPT-based Extensions

In the single-datacenter scenario - or more specifically single-server-single-queue with job preemption scenario - it has been shown that Shortest-Remaining-Processing-Time (SRPT) minimizes the average job completion time[9, 30, 31] by selecting the job with smallest remaining size first.

To the best of our knowledge, the problem of scheduling jobs across multiple datacenters has not been solved nor extensively studied. It is natural to consider SRPT-based extensions to multi-datacenter environment, as we will present next. However, we illustrate in Section 2.3.3 their shortcomings as the motivation for better approaches.

2.3.1 Global-SRPT

The first heuristic is to run the SRPT in a coordinated manner, which performs SRPT and computes the jobs priority based on the jobs' total remaining size across all the datacenters. We call this heuristic as *Global-SRPT*. Glogbal-SRPT runs at the central controller, as it requires the global state of the current jobs' remaining tasks across all the datacenters. Then central controller passes the job order computed by Global-SRPT to all the datacenters, where each datacenter scheduler updates its sub-jobs order in the queue based on the new job order.

In our motivating example, the total remaining tasks for Job A, B, C are $12, 11, 13$, respectively, so the job order computed by Global-SRPT is $B \rightarrow A \rightarrow C$, which is enforced by each datacenter as shown in Figure 2(b). Since Global-SRPT gives higher priority to the jobs with fewer tasks and finishes them as quickly as possible, it avoids the cases that small jobs are blocked behind the large jobs and spend lots of time waiting. As a result, Global-SRPT achieves better average job finish instant ($\frac{37}{3}$ as in the example) compared to that of the default scheduling FCFS (13 as in the example).

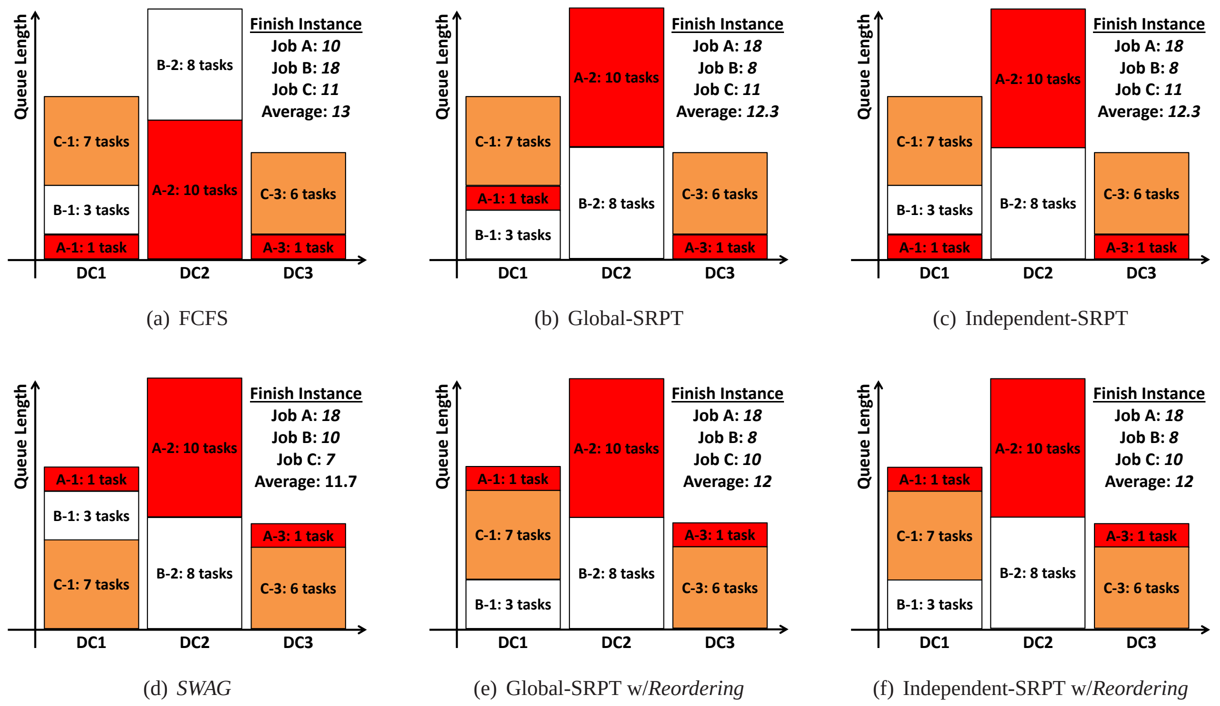
2.3.2 Independent-SRPT

Since SRPT is designed for single-scheduler scenario, our second heuristic is to enable each datacenter scheduler to perform SRPT on its own, with the hope that each datacenter reduces average completion time for its sub-jobs. We call this *Independent-SRPT*, as the datacenter prioritizes its sub-

[6] We will discuss the assumptions that make a job's finish instant equal to its finish time in Section 3, and how our system addresses those assumptions in Section 5.

Job ID	Arrival Sequence	Remaining Tasks in DC1	Remaining Tasks in DC2	Remaining Tasks in DC3	Total Remaining Tasks
A	1	1	10	1	12
B	2	3	8	0	11
C	3	7	0	6	13

Table 1. Settings of The Example: Job Set, Arrival Sequence and Task Assignment



(a) FCFS (b) Global-SRPT (c) Independent-SRPT

(d) *SWAG* (e) Global-SRPT w/*Reordering* (f) Independent-SRPT w/*Reordering*

Figure 2. Results of The Example: Job Orders and Finish Instants Computed by Different Scheduling Algorithms

jobs based on the their sizes and updates the queue order independently from the information of other datacenters.

In the example, according to the jobs' remaining number of tasks for each sub-job, their priorities at each datacenter may not be the same. In datacenter 1, the priority is $A \to B \to C$, while the priority in datacenter 2 and datacenter 3 are $B \to A$ and $A \to C$, respectively (as shown in Figure 2(c)). By reducing the finish instant of the sub-jobs in each datacenter, Independent-SRPT achieves $\frac{37}{3}$ for average job finish instant in the motivating example, which is better than FCFS (13).

2.3.3 Shortcomings of SRPT-based Extensions

Both Global-SRPT and Independent-SRPT improve the average job completion time by favoring small jobs. However, since each job may have multiple sub-jobs across all the datacenters, the imbalance of the sizes among the sub-jobs causes the problems for SRPT-based scheduling.

Take Global-SRPT for example, in Figure 2(b), we see that job A's sub-jobs in datacenter 1 and 3 finish even before its sub-job at datacenter 2 starts. Since the job's completion time is determined by the last completed sub-job across all datacenters, we can actually defer $v_{A,1}$ and $v_{A,3}$ a bit without hurting job A's finish instant, while it can yield the compute resources to the tasks of other sub-jobs, say job C in this example. The same observation is also valid for Independent-SRPT in the example, in which $v_{A,1}$ can yield to $v_{B,1}$ and $v_{C,1}$ in datacenter 1, and $v_{A,3}$ can yield to $v_{C,3}$ in datacenter 3, without delaying job A's finish instant as depicted by Figure 2(f).

As illustrated in the above example, both Independent-SRPT and Global-SRPT leave significant room for improvement as they waste resources in serving some sub-jobs while their counterparts at other datacenters are delayed due to imbalanced job execution. Next, we first propose a mechanism in Section 3 to improve the result of scheduling by eliminating the waste of resources in imbalanced job execution. Then we develop a new scheduling solution in Section 4 that leads to further improved scheduling results.

3. Reordering-based Approach

Recall that one insight into why the SRPT-based heuristics do not result in better performance is that they fail to consider the competition for resources faced by each of its component sub-jobs, as only the "slowest" sub-job determines the response time of the job. Consequently, there is no gain from lowering the response time of a sub-job at datacenter d if it has a counterpart at datacenter j with a higher completion time. In that case, we might as well delay this sub-job, in favor of other sub-jobs at datacenter d which have "faster" counterparts at other datacenters. This brings us to the notion of reordering the sub-jobs for the jobs, in a coordinated manner, based on how the sub-jobs of a job are progressing at various datacenters.

Specifically, we develop *Reordering*, as an auxiliary mechanism to reduce the "imbalance" (in terms of their position in the local queues) of a job's sub-jobs. *Reordering* can work as an "add-on" to any scheduling solution. The basic idea behind *Reordering* is to continue moving sub-jobs later in a local queue, as long as delaying them does not increase the overall completion time of the job to which they belong; this, in turn, gives other jobs an opportunity for a shorter completion time.

Algorithm 1 presents the pseudo code of *Reordering*, and its actual mechanism works as follows. Given the sub-jobs' queue order, as computed by any scheduling algorithm, in each iteration *Reordering* starts by identifying the datacenter *targetDC* with the longest queue length (Step 5) and targets the last sub-job $targetJob$ in its queue, which has the maximum value of $i_{targetJob,targetDC}$ in the queue (Step 6). We add $targetJob$ to N (Step 7), which is a queue data structure that keeps the sequence of its elements based on their arrival, and extract all of the sub-jobs associated with Job $targetJob$ from the corresponding datacenter (Step 8). The same procedure continues until all current jobs in the system have been added into N (Step 9). The final job order computed by *Reordering* is the reverse order of N (Step 10).

2 *Reordering* Algorithm

```
1:  procedure REORDERING($i_{j,d}, \forall j \in J, d \in D$)
2:      $U \leftarrow J$
3:      $N \leftarrow \emptyset$ // an ordered list
4:      while $U \neq \emptyset$ do
5:          $targetDC \leftarrow \max_d |q_d|, \forall d \in D$
6:          $targetJob \leftarrow \max_j i_{j,targetDC}, \forall j \in J$
7:          $N.push\_back(targetJob)$
8:          $q_d \leftarrow q_d - |v_{targetJob,d}|, \forall d \in D$
9:          $U \leftarrow U - \{targetJob\}$
10:     return reverse($N$)
```

In our example in Figure 2, *Reordering* improves both Global-SRPT and Independent-SRPT by delaying $v_{A,1}$ and $v_{A,3}$ until the end of their associated queues after identifying that DC2 has the longest queue length and sub-job $v_{A,2}$ is the last one in its queue. The delay of $v_{A,1}$ and $v_{A,3}$ does not degrade Job A's finish instant as it is determined by $v_{A,2}$. This procedure continues by selecting Job C, and finally Job B, which results in $N = A \rightarrow C \rightarrow B$. Thus, Reordering returns $B \rightarrow C \rightarrow A$, with a mean job finish instant of 12 for both Global-SRPT with *Reordering* and Independent-SRPT with *Reordering*, as opposed to that of $\frac{37}{3}$ without *Reordering*.

Note that in the *Reordering* algorithm, we use a job's finish instant to approximate its job finish time. Moreover, the job finish instant is exactly the job finish time under the following assumptions: (1) *homogeneous task service times*, i.e., all tasks of all jobs have the same duration; (2) *homogeneous service rates*, i.e., all servers in all datacenters serve tasks at the same rate; and (3) *homogeneous data centers*, i.e., all datacenters have an equal numbers of computing slots with the same configurations.

Under the above stated assumptions, *Reordering* would, at the very least, not result in degradation in completion time.

Theorem 1: *Reordering provides non-decreasing performance improvement for any scheduling algorithm.*

Let f_x be job x's finish instant represented by the queue position; that is, $f_x = \max_{y \in D} i_{x,y}$. Let O to be any scheduling algorithm applied to the datacenters and h_O be the resulting overall job finish instant; that is, $h_O = \frac{1}{|J|} \times \sum_{x \in J} f_x$. Let R denote the *Reordering* algorithm and $h_{O,R}$ to be the overall job finish instant of executing algorithm O and algorithm R sequentially. Theorem 1 states that $h_{O,R} \leq h_O$ no matter what scheduling algorithm O is.

Proof: We provide an intuitive proof based on Mathematical Induction on the number of jobs. When $n = 1$, the theorem obviously holds. Assume the theorem holds when $n = k$. We define $h(k)$ as the overall job finish instant when the number of jobs is k. So, $h_{O,R}(k) \leq h_O(k)$. When $n = k+1$, suppose we first process job a, since it is identified from the data-center with the longest queue, after being processed, its finish time $f_a^{'}$ is the same as f_a, which is job a's finish instant before applying *Reordering*. For the other jobs, based on step 3 we know that $h_{O,R}(k) \leq h_O(k)$. Therefore, $h_{O,R}(k+1) = \frac{k \times h_{O,R}(k) + f_a^{'}}{k+1} \leq \frac{k \times h_O(k) + f_a}{k+1} = h_O(k+1)$.

The above Theorem proves that *Reordering* improves, or does no harm at least, the average job finish instant for any job scheduling algorithm. With the assumption that job finish instant can estimate the job finish time, *Reordering* improves the average job finish time, and the average job response time as the result. In Section 5 we discuss how we address these assumptions for a system prototype, and evaluate it in Section 6.

In summary, we emphasize that *Reordering* is an add-on mechanism that can be easily used with any scheduling approach to improve (or at the very least not harm) overall average job completion time. We leave further discussions about *Reordering*'s usages until Section 8.

4. Workload-aware Approach

Given the "do no harm" property of *Reordering* as described above, it is naturally a conservative approach (to modifying the original scheduling decisions), with results depending significantly on the original scheduling algorithm to which the reordering process is applied. However, *Reordering* still leaves rooms for improvement. In the motivating example in Section 2, both Global-SRPT (Figure 2(e)) and Independent-SRPT (Figure 2(f)) came up with the job order of $B \rightarrow C \rightarrow A$. We observe that the scheduling performance would be improved if we switched the order of job B and job C, and result in the new job order $C \rightarrow B \rightarrow A$. Doing so would bring performance improvement for job C while hurting the completion time of job B, which is against the principle of Reordering, yet the net effect results in overall performance improvement as shown in Figure 2(d). This observation motivates us to develop the more aggressive approach than *Reordering*, termed *Workload-Aware Greedy Scheduling (SWAG)*, which schedules the jobs greedily based on their estimated finish time. We first discuss the design principles for *SWAG*, and then present its algorithm details.

4.1 SWAG Design Principles

Recall that a job's completion time is composed of the waiting time as well as the service time, and the traditional SRPT results in the shortest total waiting time for all jobs by greedily scheduling the job with the shortest remaining processing time over the long ones. Therefore SRPT optimizes the average job completion time since the jobs' service times are fixed. [7] This insight is common for all job scheduling in reducing the average job completion time, yet it sets the ground of our first design principle for *SWAG*.

First Principle: *In order to reduce the total waiting time and further reduce the response time, jobs that can finish quickly should be scheduled before the other jobs.*

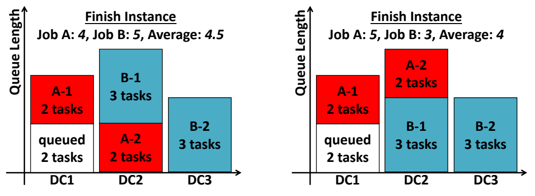
However, as shown in Section 2.3.3, following the first principle by favoring the small jobs only is sub-optimal in the multiple-scheduler-multiple-queue scenario, due to the imbalance between the sizes of the sub-jobs across the datacenters and the fact that the finish time of a job depends only on its last completed sub-job. In fact, a small job with a large sub-job may not finish as quickly as a large job with many small sub-jobs. Therefore it leads us to the second design principle.

Second Principle: *Since the small jobs are not guaranteed to finish quickly (as is the case in the single-scheduler-single-queue scenario), we should consider scheduling jobs more as a function of sub-job sizes rather than the size of the overall job.*

The first two principles guide us to select the job finishing the quickest under the condition that it occupies the entire system. However each datacenter has different workload at the scheduling decision instant, which also impacts the waiting time that each sub-job suffers. This gives us the final design principle for *SWAG*.

Third Principle: *Since the sub-jobs of a job experience different delays at different datacenters, we should also consider the local queue sizes in assessing the finish times of sub-jobs.*

Figure 3 presents a simple example to illustrate the third principle, in which there are two jobs A and B to be scheduled over 3 datacenters, and there are two tasks already at the first datacenter. Note that both Global-SRPT and Independent-SRPT would result in the scheduling result shown in Figure 3(a) as they both prioritize the jobs or sub-jobs based on their sizes only. Also note that executing *Reordering* after Global-/Independent-SRPT does not im-



(a) SRPT-based Approach (b) Better Approach

Figure 3. Motivating Example for Third Principle

prove their performances because the dominating jobs and sub-jobs are already put at the end of the queue.

In conclusion, all the 3 principles are essential for reducing the average job completion time. Next, we present how we construct *SWAG* based on these principles.

4.2 SWAG Algorithm

In our design, the central controller runs *SWAG* whenever a new job arrives or departs. The new order of all jobs is computed from scratch based on the estimated job finish times. Let q_d denote the current queue length at datacenter d, and $|v_{j,d}|$ denote the size of job j's sub-job at datacenter d. $v_{j,d} = 0$ if none of job j's tasks is assigned to datacenter d. In addition, we define the *makespan* m_j for job j as: $m_j = \max_d(q_d + |v_{j,d}|), \forall d \in D$.

Then *SWAG*—as detailed in Algorithm 3—greedily prioritizes jobs by computing their estimated finish times based on the current queue length (accumulated number of tasks to be served) as well as the job's remaining size (number of remaining tasks). Initially, *SWAG* computes the makespan for each job based on Equation **??** (Step 5). Then *SWAG* selects the job with the minimal makespan (Step 6), appends it into the job order (Step 7) and updates the queue length based on the selected job's sub-job sizes (Step 8). If there are more than one job with the minimal makespan, *SWAG* picks the one with the smallest total remaining size as a tie-breaker. *SWAG* continues to greedily add the next job with the smallest makespan, with respect to the current queue lengths, until all the current jobs in the system have been added.

4 Workload-Aware Greedy Scheduling (*SWAG*)

```
1: procedure SWAG($J, v_{j,d}, \forall j \in J, d \in D$)
2:     $N \leftarrow \emptyset$ // an ordered list
3:     $q_d \leftarrow 0, \forall d \in D$
4:     while $|N| \neq |J|$ do
5:         $m_j \leftarrow \max_d (q_d + |v_{j,d}|), \forall j \in J, d \in D$
6:         $targetJob \leftarrow \min_j m_j, \forall j \in J$
7:         $N.push\_back(targetJob)$
8:         $q_d \leftarrow q_d + |v_{targetJob,d}|, \forall d \in D$
9:     return $N$
```

In our example presented in Figure 2, *SWAG* first selects Job C as it has the smallest makespan of 7, as compared to 10 for Job A and 8 for Job B. After that, the queue length for datacenter 1 and datacenter 3 would be updated to 7 and 6, respectively, according to Job C's sub-job size. At this point, both Jobs A and B result in the same makespan of 10, with

[7] Note that in a traditional scheduling problems a job is an atomic processing unit, as opposed to our problem where a job is composed of small tasks that can be executed in parallel.

respect to the new queue lengths. Since Job B has a smaller remaining size than Job A, it is added after Job C, followed by Job A . The final job order as computed by SWAG is $C \rightarrow B \rightarrow A$, and the resulting average job finish time is $\frac{35}{3}$, which is better than that the SRPT-based solutions.

5. Prototype and System Extensions

In this section we describe our prototype implementation and how we address some system issues.

Prototype: We implemented a system prototype with Spark[39]. Two main components in our system prototype are the global controller and the local controller.

The global controller is primarily in charge of computing the job orders, by running *Reordering* or *SWAG* module, based on the information (e.g., number of remaining tasks of each job at each datacenter) collected from each local controller. The global controller passes the results of job orders to each local controller through socket communication. Besides, whenever a new job arrives, it divides the job into sub-jobs and send the metadata (e.g., the application program ID, the number of tasks) to each local controller.

The local controller is in charge of feeding the computed job orders to the local cluster as well as reporting the jobs' progress to the global controller. Based on the updated job order, each cluster scheduler assigns the next available computing slots to the tasks of the job with the highest priority until all of its tasks are launched. In addition to passing new job orders to the cluster, the local controller sends the global controller updates of jobs' progress (e.g., number of finished tasks for each job), upon receiving requests from the global controller by reading the logs produced by Spark cluster.

Heterogeneous Datacenter Capacity. In previous sections we assume all datacenters to be homogeneous in that they have the same number of computing slots for serving the tasks. In reality datacenters may have different capacity in the number of computing slots. Recall that both *Reordering* and *SWAG* rely on queue length as the estimation for job finish time (e.g., Step 5 in Algorithm 1 and Step 5 in Algorithm 3), while the same queue length would result in different job finish time if equipped with different number of computing slots. *Reordering* and *SWAG* can easily adapt to heterogeneous datacenter capacity by normalizing the queue length of each datacenter by their number of computing slots. For example, Step 5 in Algorithm 1 can be updated as $targetDC \leftarrow \max_d [\frac{|q_d|}{c_d}], \forall d \in D$, and Step 5 in Algorithm 3 can be updated as $m_j \leftarrow \max_d [\frac{(q_d+|v_{j,d}|)}{c_d}], \forall j \in J, d \in D$, where c_d represents the number of computing slots in datacenter d. The intuition is that the datacenters with more computing slots spend shorter time finishing serving the same workload than the datacenters with less computing slots.

Heterogeneous Tasks Duration. In above presentation we assumed that all tasks across all jobs were of the same duration. However, previous works [5, 7, 8] show that tasks duration could be heterogeneous within and across jobs in a real system due to various reasons. We address this by having the local scheduler of each datacenter select the task with the longest expected duration that is not yet launched for the sub-job with the highest priority determined by the job scheduling. The rationale behind this method is to start the larger tasks earlier in order to reduce the makespan across all tasks of a particular sub-job.

Inaccuracies in Task Duration Estimation. The way we address heterogeneous tasks duration (task-level scheduling by local schedulers) relies on reasonably accurate estimation of task durations. Unfortunately, there is no guarantee that the estimations at the scheduler are accurate because tasks duration are subject to many dynamic factors[5, 7, 8], including I/O congestion and performance interference among concurrent tasks. The typical approach to this problem is to use the finished tasks' duration to estimate the duration of the remaining tasks of the same job[5, 7, 8]; it is reported that the estimation accuracy with such approaches reaches $\approx 80\%$, as the jobs get closer to the completion[8]. Here, we do not assume a specific estimation mechanism for task duration, but rather (in Section 6) evaluate the sensitivity of our system's performance to the estimation accuracy.

Scheduling Decision Points. The heterogeneous nature of tasks duration and the (potential) lack of accuracy in their estimation indicate that in a real system we should consider (re)evaluating scheduling decisions at task departure points (in addition to job arrival and departure points). However, our simulation study indicates that the heterogeneous nature of tasks duration and the inaccuracies in their estimation only have a marginal impact on the scheduling results. Since the frequency of task departures can be a few orders of magnitude larger than that of job arrivals and departures, running of job-level scheduling at such high frequency would incur substantial overhead, particularly as job-level scheduling is performed by the central controller. Consequently, we conclude that in a real system it is sufficient to consider scheduling decisions upon job arrivals and departures.

6. Performance Evaluation

In this Section we conduct an extensive simulation study, with realistic job traces, for the proposed scheduling approaches (*SWAG* and *Reordering*) compared to the traditional solutions (FCFS and SRPT extensions) with regard to performance improvement and fairness (Section 6.2), overhead evaluation (Section 6.3) and sensitivity analysis (Section 6.4). Our results show that *SWAG* and *Reordering* improve SRPT-based approaches by 50% and 27%, respectively, over a wide range of settings.

6.1 Experiment Settings

The main performance metric we focus on is average job completion time, which is defined as the average elapsed duration from the job's arrival time to the time instant at which the job has all its tasks completed and can depart from the system. Average job completion time is a common

Trace Type	Avg. Job Size	Small Jobs $(1-150$ tasks)	Medium Jobs $(151-500$ tasks)	Large Jobs (501+ tasks)	Trace Characteristic
Facebook[5–8]	364.6 tasks	89%	8%	3%	high variance with a few extremely large jobs
Google[2, 28, 33]	86.9 tasks	96%	2%	2%	small variance with a few large jobs
Exponential	800 tasks	18%	29%	53%	moderate variance in job sizes

Table 2. Job Traces

metric for data analytics systems; this is a reasonable metric when focusing on customer quality-of-service. In addition, we also evaluate the jobs' *slowdown*, which is defined as the job completion time divided by the job service time. We use slowdown as a metric for evaluating fairness among jobs of different sizes, as detailed in Section 6.2. All performance results are presented with confidence intervals of $95\% \pm 5\%$.

We compare the performance of: FCFS, Global-SRPT, Independent-SRPT, Global-SRPT followed by *Reordering*, Independent-SRPT followed by *Reordering*, and *SWAG*. We also show the results generated by Optimal Scheduling, which are obtained through an offline brute-force search, i.e., with full knowledge of future job arrivals and actual tasks duration. We use the results from Optimal Scheduling as an upper-bound on the response time improvement that can be achieved through better scheduling, to investigate how much room for improvement is left. We run FCFS as our baseline scheduling approach, for comparison purposes only. For clarity of exposition, we present our results as the normalized average job completion time of each algorithm, i.e., normalized by the average job completion time achieved by the FCFS approach for the same setting.

Workload: We use synthetic workloads in our experiments with job size distributions obtained from Facebook's production Hadoop cluster [5–8] and Google cluster workload trace[2, 28, 33], as well as the Exponential Distributions, referred to as Facebook trace, Google trace and Exponential trace, respectively. Table 2 summarizes the job traces we use in our simulation experiments. We adjust the jobs' inter-arrival times for both workloads based on Poisson Process in order to make the two workloads consistent in terms of system utilization. The default settings for the average job size is 800 tasks, and we tune the inter-job-arrival time to obtain the workload with certain system utilization.

Tasks Duration: The tasks duration in our simulations are modeled by Pareto distribution with $\beta = 1.259$ according to the Facebook workload information described in [8], and average task duration to be 2 seconds. In our simulation experiments, we investigate the impact of inaccurate estimation of task duration in Section 6.4.

Task Assignment: To evaluate the impact of imbalance due to task assignment, we use Zipf Distribution to model the skewness of task assignment among the datacenters. The higher the Zipf's skew parameter is, the more skewed that tasks assignment is (i.e., constrained to fewer datacenters). We also consider two extreme cases where tasks of each job are: (i) distributed uniformly across all datacenters, or (ii) assigned to only one datacenter. The default setting for the skew parameter is 2, while we investigate how skew of task assignment affects the performance in Section 6.4.

System Utilization: We define the percentage of occupied computing slots as our system utilization. Multiple factors contribute to the system's utilization: job inter-arrival time, job size, task duration, and task assignment.

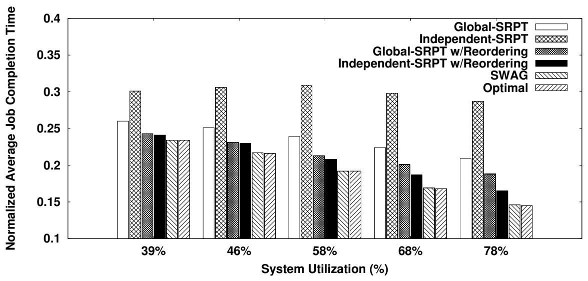
Other Default Settings: In our experiments the default number of datacenters is 30, with 300 computing slots per datacenter. Such default system settings result in $\approx 78\%$ system utilization, which allows us to explore how the system performance behaves at reasonably high utilization.

6.2 Scheduling Performance Results

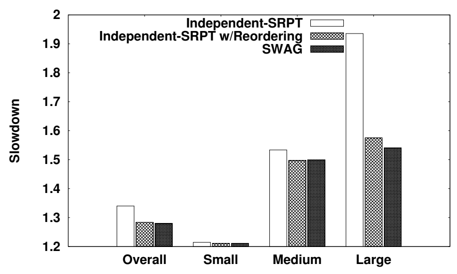
Figure 4(a), 4(c) and 4(e) depict the average job completion time (normalized by that of FCFS), using the Facebook trace, Google trace and Exponential trace respectively. We vary the average job inter-arrival times and observe how performance characteristics react to different system utilization.

Performance Improvements of Reordering Our experiment results first confirm that *Reordering* does result in reduction of average completion time for SRPT-based heuristics, as stated by *Theorem 1*. The performance improvements for SRPT-based heuristics due to *Reordering* reaches as high as 27% under highly utilized settings, and is up to 17% under lower utilization. Finally, the results also show that *Reordering* is more beneficial to Independent-SRPT than to Global-SRPT. This is intuitive as Independent-SRPT does not coordinate between the sub-jobs of a job and thus results in a higher imbalance between the sub-jobs; this creates more opportunities for *Reordering* to improve performance.

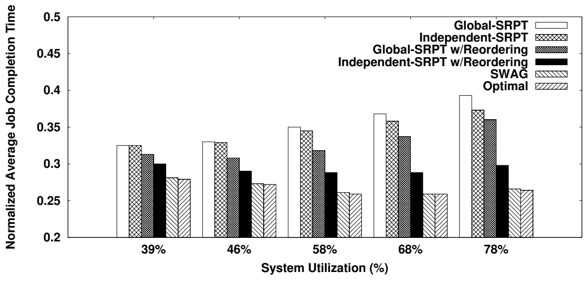
Without *Reordering*, Global-SRPT performs better than Independent-SRPT in the Facebook trace, while the Google trace and the Exponential trace display the opposite trend. Under higher utilization, Global-SRPT outperforms Independent-SRPT by 27% in the Facebook trace, while in Exponential trace, Independent-SRPT outperforms Global-SRPT by 32%. This is the result of the fact that the variance of job sizes in the Facebook trace is significantly higher than that of the Google trace and the Exponential trace, so Global-SRPT benefits more from favoring small jobs by considering the total job size across all datacenters, while Independent-SRPT performs even poorly by considering only the individual sub-job sizes. In the Google trace, however, the gap between Global-SRPT and Independent-SRPT is not obvious. Most of the jobs in Google trace are small and so are



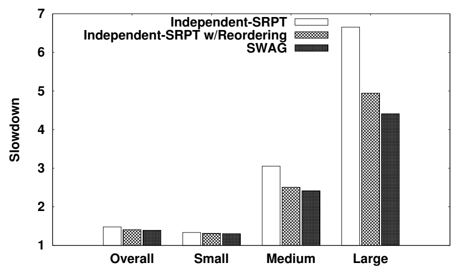
(a) Performance with Facebook Trace



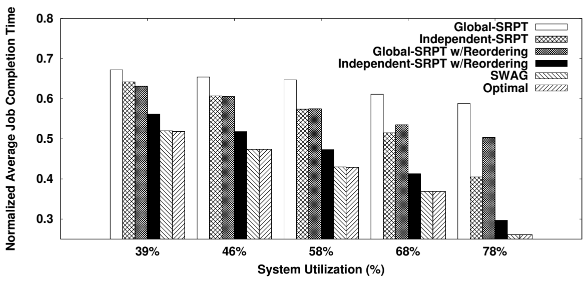
(b) Fairness with Facebook Trace



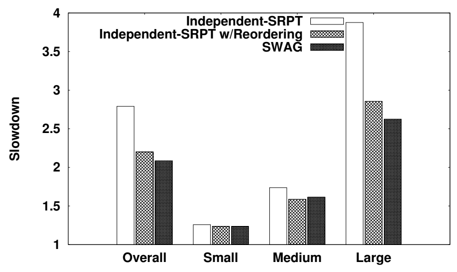
(c) Performance with Google Trace



(d) Fairness with Google Trace



(e) Performance with Exponential Trace



(f) Fairness with Exponential Trace

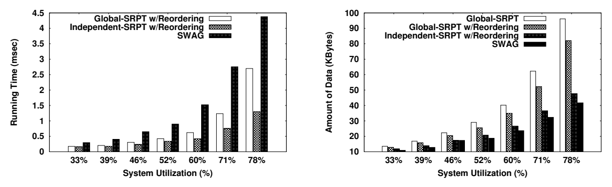
Figure 4. Performance and Fairness Results with Different Workload Traces

the variance in the job sizes. With such characteristic, the skews among the sub-job sizes tend to be smaller compared to the other two job traces, and, therefore, Global-SRPT and Independent-SRPT perform similar job scheduling decision.

With *Reordering*, Independent-SRPT performs better than Global-SRPT in all traces, because Independent-SRPT benefits significantly from *Reordering* than Global-SRPT does as mentioned above. The gap between them becomes significant (10% or more) starting at lower utilization (39%) in Exponential trace, and reaches 40% under higher utilization. In the Facebook trace, however, the gap is only significant under higher utilization (68% and 78%). This is because Global-SRPT performs reasonably well, unlike Independent-SRPT without *Reordering*, in the Facebook trace. Thus, Global-SRPT with *Reordering* also performs well as compared to the performance in the Exponential trace. These results also show that the performance of *Reordering* depends on the original scheduling algorithm.

Performance Improvements of *SWAG* Compared to SRPT-based heuristics, *SWAG*'s performance improvements under higher utilization are up to 50%, 29% and 35% in the Facebook, Google and Exponential trace respectively, with at least 12% improvement under lower utilization. The differences in performance improvements attribute to the fact that job traces with higher variance in job sizes tend to have more large jobs, which potentially results in more sever skews among the sub-jobs. Thus, high-variance job trace like Facebook trace displays more opportunities that allow *SWAG* to achieve higher improvement by selecting jobs that can finish quickly according to its design principles. In addition, *SWAG* outperforms, by up to 10%, SRPT-based heuristics with *Reordering*, under various utilization and in all job traces. Finally, *SWAG* achieves near-optimal performance throughout our experiments: the performance gap between *SWAG* and Optimal is within only 2%.

Fairness among Job Types: Figure 4(b), 4(d) and 4(f) present the slowdown results for the Facebook, Google and Exponential trace respectively. We further present the slowdown for different job types by classifying the jobs based on their sizes (number of tasks): small jobs (1-150 tasks), medium jobs (151-500 tasks) and large jobs (501 or more tasks). The slowdown for FCFS is omitted as it is significantly larger than the rest and is more than 15 in all cases. Also, Global-SRPT and Independent-SRPT have similar results; thus, we only include the results for one of them.



(a) Scheduling Running Time (b) Communication Overhead

Figure 5. Scheduling Overhead Results

We note that that all scheduling approaches have the same trends, i.e., that small jobs have the smallest slowdown while large jobs have the largest slowdown. As expected, this is naturally due to the fact that all algorithms essentially favor smaller jobs in order to reduce the average job completion time. In addition, the major differences of slowdown between the scheduling solutions exist in large jobs.

In Facebook and Exponential trace, the slowdown of large jobs for Independent-SRPT is 40% more than its overall slowdown, while the gap is no more than 30% for Independent-SRPT with *Reordering* and no more than 25% for *SWAG*. Google trace displays significant gap of slowdown between large jobs and overall jobs. This is because most of the jobs in Google trace are small jobs, therefore the few large jobs are often queued for a long time while the system is serving many small jobs as determined by the scheduling solutions. However, Independent-SRPT with *Reordering* and *SWAG* still maintain relatively low slowdown compared to Independent-SRPT. Hence, we conclude that *Reordering* and *SWAG* improve performance without significantly sacrificing performance of large jobs.

We also observe that *Reordering* improves the original scheduling approach by mainly improving performance of large jobs. This is because small jobs get to be served earlier than the other even after *Reordering* is performed, while *Reordering* provides the opportunity for some large jobs to get served earlier by delaying some other sub-jobs.

We use the Exponential trace for the following overhead and sensitivity evaluation as it displays moderate characteristics compared to the other two.

6.3 Overhead Evaluation

We evaluate our system overhead on the following aspects.

Computational Overhead. We obtain this by monitoring the execution time due to running of the scheduling algorithms during each scheduling decision point.

Communication Overhead. This is defined as the additional messages required by the global scheduler as needed to be transferred from each local datacenter to the central controller. Note that this does not include the fundamental and necessary information needed by the system, e.g., the metadata of the jobs and the tasks, or the task program binaries. Instead, It includes the information such as the set of current jobs IDs as well as their remaining number of tasks associated at each datacenter.

Figure 5(a) depicts the scheduling running time under various system utilization. The results for FCFS, Global-SRPT and Independent-SRPT are omitted as they are negligible compared to the rest. These results suggest that even under higher utilization (78%), the scheduling running time of *SWAG* (4.5*ms*) is relatively small compared to the average task duration time (2*s*). In addition to the scheduling running time, our prototype confirms that the control message passing between the global scheduler and the local scheduler required by *Reordering* and *SWAG* takes no more than a few hundred milliseconds. As a result, the delay in scheduling running time and message passing does not significantly degrade the completion time of the jobs. Note that although *SWAG* has a higher computational (worst-case) complexity than *Reordering* ($O(n^2 \times m)$ for *SWAG* and $O(n \times m)$ for *Reordering*, where n is the number of current jobs and m is the number of datacenters), the actual difference in computational overhead between *SWAG* and SRPT-based heuristics with *Reordering* is not significant, because *SWAG* is able to keep the number of current jobs (i.e., n) in the system small, by scheduling jobs that can finish quickly.

Figure 5(b) depicts the communication overhead incurred by each scheduling algorithm. Note that FCFS and Independent-SRPT do not require any additional information from local schedulers, so their overhead is zero. The communication overhead essentially depends on the number of current jobs in the system. Since *SWAG* succeeds in keeping the number of current jobs small, it achieves the smallest communication overhead.

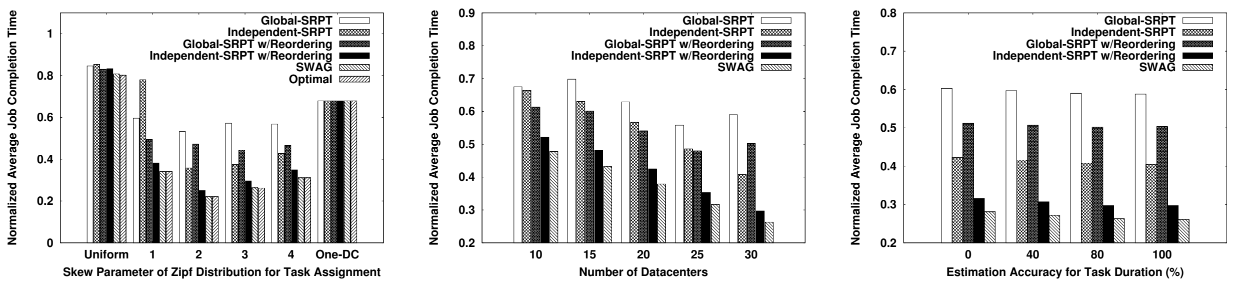
The overhead analysis confirm that the performance gains from the proposed *Reordering* and *SWAG* techniques come with acceptable computation and communication overhead.

6.4 Performance Sensitivity Analysis

Impact of Task Assignment In this experiment we study the sensitivity of scheduling algorithms to the skew in task assignments. In Figure 6(a), the X-axis represents the skewness of task assignment, with Uniform Distribution being the least skewed and One-DC Assignment being the most skewed. Between Uniform and One-DC are the results under different Zipf's skew parameters.

The general trend in Figure 6(a) is that as the skewness increases, the performance of the scheduling algorithms first increases and then decreases. There is not much room for improvement when all tasks are uniformly distributed across datacenters. The performance improvement becomes more significant as the imbalance in task assignments requires greater coordination of jobs scheduling across the datacenters to reduce the jobs' completion time. Beyond a certain skewness level, the imbalance of task assignment becomes so substantial that most of the tasks from the same job only span a few datacenters, in which case not much can be done.

As expected, when all the tasks of a job are assigned to a single datacenter, the execution of Global-SRPT and Independent-SRPT are essentially the same as they are both equivalent to performing SRPT on the local datacenters ex-



(a) Various Task Assignment Scenarios (b) Various Number of Datacenters (c) Various Estimation Accuracy

Figure 6. Performance Sensitivity Results

clusively. In this case, there is no room for *Reordering* to improve SRPT-based approach either.

Among the scheduling algorithms, *SWAG* and Independent-SRPT are more sensitive to the changes in skewness of task assignment than Gobal-SRPT. This is because their scheduling decisions are subject to how the sub-jobs of the other jobs are ordered at each datacenter, which is directly impacted by the extent of skews among the sub-jobs of the same job. On the other hand, Global-SRPT considers only the global view of the job sizes across all the datacenters, and is therefore less sensitive to how the skewness varies.

Number of Datacenters In this experiment we investigate how the number of datacenters affects the performance by varying the number of datacenters while keeping the total number of computing slots constant. In Figure 6(b), the performance improvements by *Reordering* and *SWAG* generally increase as the number of datacenters increases, because more datacenters provide greater opportunities for coordination of sub-jobs across the datacenters.

Accuracy of Task Duration Estimation In this experiment we study how the error in task duration estimation affects the scheduling algorithms' results. The estimation error happens as task execution is subject to unpredictable factors like I/O congestion and interference as discussed in Section 5, and it has impact on how local schedulers schedule the tasks because the scheduling decisions are based on estimates of tasks duration. We introduce estimation error to our experiments based on a uniform distribution with the original task duration as the average. For example, if we want to investigate 75% estimation accuracy, we set the estimation value for task duration to be uniformly drawn from the range of $[0.75, 1.25] * actual\ taskduration$, so that the estimation error is at most 25% of the actual task duration.

Figure 6(c) shows that the performance improves marginally as the estimation accuracy for task duration increases. This is because there is often a high variance in task duration due to stragglers [5, 7, 8], and the estimation error is not significant enough to affect the order of task scheduling much. Therefore, our *Reordering* and *SWAG* algorithms are robust to estimation errors.

7. Related Works

Distributed Job Execution. Relatively little work exists in the research literature on running applications on geo-distributed datacenters [14, 27, 35, 36]. Dealer [14] *dynamically* redistributes poorly performing tasks of a single job to other datacenters to reduce user-perceived latency. However, it only uses one datacenter at a time per job phase. In contrast, our work distributes tasks of a job (upon arrival) among multiple datacenters, based on data locality, with the advantage of reducing resource (network, storage) usage.

JetStream [27] focuses on the scenario in which applications aggregate data across wide-area networks, and deals with insufficient backhaul bandwidth by applying pre-processing at each source site before transferring all data to the central location. In addition to having bandwidth overhead reduction advantages, our work also dynamically adjusts scheduling decisions across datacenters, to further reduce average job completion time.

The closest works to ours are [35, 36], which propose to push the analytical queries to where the data are hosted and optimize their execution plans accordingly. Another work [26] similar to ours further improves bandwidth usage of geo-distributed analytics by placing data and computation across datacenters based on their bandwidth constraints. None of the above mentioned works address the challenges of job scheduling in distributed job execution setting; to the best of our knowledge, our work is the first to address the job-level scheduling problem in multi-datacenters.

Data Locality Scheduling. Scheduling jobs and tasks to meet data locality within clusters has been a recent trend [18, 34, 38] as it significantly improves average job completion time. Our work focuses on job scheduling given the task distribution among the datacenters, and therefore is orthogonal to the above-mentioned works as their approaches to scheduling for data locality within a datacenter can be combined with our solution.

Conventional Job Scheduling. Shortest Remaining Processing Time (SRPT) is a well-known scheduling algorithm that achieves optimal average job completion time for preemptive job scheduling in a single-server-single-queue environment [9, 30, 31]; it has been extensively studied and applied to many problem domains [15, 21, 22, 32, 37]. Our problem setting, specifically job scheduling in distributed job execution scenarios, differs in that: (a) jobs are composed of tasks that can run in parallel, and (b) tasks of the same job potentially span multiple datacenters, each with a number of compute slots, controlled by a local scheduler. As shown in Section 2.3, SRPT-based extensions do not work well in this context, mainly due to skew in a sub-job's task distribution. Several efforts, in a more idealized (theoretical) settings, include *concurrent open shop* problems [12, 23, 29], in which each job has certain operations to be processed at each machine, and the goal is to minimize the weighted average job completion time. Our work differs from that of *concurrent open shop* [12, 23, 29] in several ways: (a) we address a more general scheduling problem as each datacenter (or, machine as termed in *concurrent open shop*) has multiple compute slots that can run the tasks of the same job in parallel, (b) we develop online scheduling mechanisms rather than the offline deterministic scheduling analysis proposed by previous efforts on *concurrent open shop*, and (c) we conduct simulation-based experiments to evaluate the performance of scheduling solutions under more realistic settings and with realistic workloads.

Coflow Scheduling Varys [10] addresses a similar scheduling problem; it schedules coflows, each composed of several sub-flows. Despite sharing the goal of average completion time reduction, there exist several major differences in our problem settings. In addition to the coflows' sizes and the current workloads, the scheduling results of Varys depend on sending and receiving rates at the two ends of a flow, while there is no such constraint in our problem setting. Another difference is that Varys schedules the coflows and their sub-flows all at the central controller, while in our case, scheduling of jobs and tasks is carried out through the collaboration of global and local schedulers.

8. Discussions

SWAG vs. Reordering The two approaches proposed in this paper are *Reordering* and *SWAG*. *Reordering* is a lightweight add-on that can be easily combined with any scheduling approach, potentially improving average job completion time, while *SWAG* is a stand-alone scheduling approach. Both incur the same communication overhead in collecting information from local datacenter schedulers, to support global job scheduling decisions. Although *SWAG* has a greater computational overhead than *Reordering* (as shown in Section 6.3), our simulation results (in Sections 6.2 and 6.4) show that *SWAG* outperforms all other scheduling algorithms in all settings, including those improved by *Reordering*, largely due to the principles upon which it is designed.

We note, however, that *Reordering* can easily adapt to certain job scheduling constraints that *SWAG* may not be able to address. For instance, if jobs have deadlines and deadline-aware scheduling is used, *Reordering* can still improve, or at least not harm, the average job completion time.

Finally, note that *SWAG* cannot be further improved by *Reordering*. Recall that in each iteration of *Reordering* execution, it selects the last job in the most-loaded datacenter and adds it into the final job order (in reverse). Since *SWAG* schedules jobs based on minimizing their makespan, all of the sub-jobs of the job selected by *Reordering* are already at the end of the queue of their associated datacenter. Therefore, applying *Reordering* after *SWAG* returns exactly the same job order as that obtained by running *SWAG* only.

DAG of Tasks. Real workloads suggest that jobs can typically be modeled as a DAG of tasks - the first-stage tasks process their input data from physical storage, and tasks in following stages aggregate the output from the first-stage tasks. Note that those following stages initiate data shuffling across the geo-distributed datacenters through wide area networks, which can incur unpredictable latency and potentially large cost at the backhaul.

Our presentation here essentially assumes jobs are composed of single-stage tasks. To extend our approach to multi-stage jobs, we can first assign the first-stage tasks to the datacenters hosting the input data (using the algorithms proposed here), then redirect the tasks of the following stages and transfer all the intermediate results from the first-stage tasks to the datacenter that has the largest sub-job of the original job. The tasks of the following stages can then run within this datacenter and the sizes for all the original sub-jobs can be updated accordingly. As a result, there remains a single sub-job (of the original job) for the remainder of its stages.

Data Transfer Schedule. In running jobs across geo-distributed datacenters, a job's completion time depends on not only how the jobs are scheduled for service, but also how the data transfer flows are scheduled. To reduce the overall job completion time, our work takes the initial step of coordinated job scheduling across all datacenters. For finer control and further improvements in overall job completion time, our system can be extended to consider how data transfer flows consume wide area network bandwidth, e.g., when the flow should start sending data at what transmission rate.

Multiple Task Placement Choices. In this paper we assume each task can only be placed at the datacenter that has its required data. One future extension is to allow each task multiple placement choices which would result in a joint optimization of job scheduling and task placement.

9. Conclusions

In the big data era, as data volumes keep increasing at dramatical rates, running jobs across geo-distributed datacenters emerges as the promising trend. In this setting, we propose two solutions for job scheduling across datacenters: *Reordering*, which improves scheduling algorithms by efficiently adjusting their job order with low computational overhead; and *SWAG*, a workload-aware greedy scheduling algorithm that further improves the average job completion time and achieves near-optimal performance. Our simulations with realistic job traces and extensive scenarios show that the average job completion time improvements from *Reordering* and *SWAG* are up to 27% and 50%, respectively, as compared to SRPT-based extensions, while achieved at reasonable computational and communication overhead.

References

[1] http://aws.amazon.com/about-aws/global-infrastructure/. Amazonn Global Infrastructure.

[2] https://code.google.com/p/googleclusterdata/. Google Cluster Workload Traces.

[3] http://hadoop.apache.org/. Hadoop Cluster Computing System.

[4] http://www.microsoft.com/en-us/server-cloud/cloud-os/global-datacenters.aspx. Microsoft Cloud Platform.

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

[6] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: coordinated memory caching for parallel jobs. In *USENIX NSDI*, 2012.

[7] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *USENIX NSDI*, 2013.

[8] G. Ananthanarayanan, C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: trimming stragglers in approximation analytics. In *USENIX NSDI*, 2014.

[9] N. Bansal and M. Harchol-Balter. *Analysis of SRPT scheduling: Investigating unfairness*. ACM, 2001.

[10] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM*, 2014.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 2013.

[12] N. Garg, A. Kumar, and V. Pandit. Order scheduling models: hardness and algorithms. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. Springer, 2007.

[13] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *Proceedings of the VLDB Endowment*, 2014.

[14] M. Hajjat, D. Maltz, S. Rao, K. Sripanidkulchai, et al. Dealer: application-aware request splitting for interactive cloud applications. In *ACM CoNEXT*, 2012.

[15] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems (TOCS)*, 2003.

[16] J.-H. Hwang, U. Cetintemel, and S. Zdonik. Fast and reliable stream processing over wide area networks. In *IEEE ICDE Workshop*, 2007.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, 2007.

[18] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM SOSP*, 2009.

[19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM*, 2013.

[20] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *ACM EuroSys*, 2013.

[21] M. Lin, A. Wierman, and B. Zwart. The average response time in a heavy-traffic srpt queue. *ACM SIGMETRICS Performance Evaluation Review*, 2010.

[22] M. Lin, A. Wierman, and B. Zwart. Heavy-traffic analysis of mean response time under shortest remaining processing time. *Performance Evaluation*, 2011.

[23] M. Mastrolilli, M. Queyranne, A. S. Schulz, O. Svensson, and N. A. Uhan. Minimizing the sum of weighted completion times in a concurrent open shop. *Operation Research Letter*, 2010.

[24] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook warm blob storage system. In *USENIX OSDI*, 2014.

[25] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *IEEE ICDE*, 2006.

[26] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. Low latency geo-distributed data analytics. In *ACM SIGCOMM*, 2015.

[27] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *USENIX NSDI*, 2014.

[28] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[29] T. A. Roemer. A note on the complexity of the concurrent open shop problem. Springer, 2006.

[30] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 1968.

[31] L. E. Schrage and L. W. Miller. The queue m/g/1 with the shortest remaining processing time discipline. *Operations Research*, 1966.

[32] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions on Internet Technology (TOIT)*, 2006.

[33] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011.

[34] J. Tan, X. Meng, and L. Zhang. Delay tails in mapreduce scheduling. *ACM SIGMETRICS Performance Evaluation Review*, 2012.

[35] A. Vulimiri, C. Curino, B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *To Appear in NSDI*, 2015.

[36] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese. Wanalytics: Analytics for a geo-distributed data-intensive world. In *To Appear in CIDR*, 2015.

[37] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an m/gi/1. In *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[38] 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 *ACM EuroSys*, 2010.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX NSDI*, 2012.