On Task Assignment and Scheduling for Distributed Job Execution

Yitong Guan
School of Computer Science and Engineering
Nanyang Technological University, Singapore 639798
guan0049@e.ntu.edu.sg

Xueyan Tang
School of Computer Science and Engineering
Nanyang Technological University, Singapore 639798
asxytang@ntu.edu.sg

***Abstract*—It is common for big data applications to run across multiple datacenters or machine clusters because the data inputs are distributed over different locations. This paper studies a job scheduling problem for distributed job execution in which the data inputs to jobs may be replicated across multiple locations so that each task of a job can be executed at any one of these locations. To schedule the jobs, we need to determine the processing locations for the tasks of each job and the execution order of the tasks at each location. We focus on the objective of minimizing the average job response time. We first design a task assignment algorithm to balance the task allocation among various locations. We then further develop integrated solutions that conduct task assignment and scheduling together. We experimentally evaluate our algorithms using real job traces. The results show that our algorithms can significantly reduce the job response times compared to a baseline that allocates each task to a fixed location for processing.**

I. Introduction

In contemporary big data applications, jobs often need to be executed in a distributed manner across multiple datacenters or machine clusters because the data inputs are available at different locations [1], [2]. Job scheduling plays an important role in the performance of big data processing systems that require distributed execution. When multiple jobs are running concurrently in the system, properly scheduling the processing order of jobs can make the system more efficient. Existing work on scheduling for distributed job execution [3] has assumed that the data input for each task is available at only a single datacenter or machine cluster. In practice, for reliability or fault tolerance considerations, it is common for the data to be replicated across several locations. As a result, there are multiple choices for placing tasks that preserve data locality. This creates new opportunities and challenges for scheduling distributed job execution.

In this paper, we formulate and study a job scheduling problem for distributed job execution in which the data to be processed by each job is possibly available at multiple locations. To schedule the job execution, we need to decide where to execute the tasks of each job subject to data locality requirements and in what order to execute the tasks at each location. We focus on an online setting in which new jobs are released over time and scheduling decisions have to be made on the fly. Our goal is to minimize the average job response time where the response time of a job is given by its completion time minus its release time.

We propose a number of solutions to the scheduling problem. First, we design algorithms to assign the tasks among different locations for execution. The objective of task assignment is to balance the task allocation among the locations so as to optimize job response times. On completing the task assignment, various scheduling algorithms can be used to determine the task execution order at each location. Second, we design algorithms to assign tasks and determine their execution order in an integrated fashion. This allows the task allocation to take into account the job priorities decided by the scheduling strategy. We explore different levels of task assignment adaptivity to job arrivals in the online setting and adopt heuristics to trade the quality of task allocation for scheduling efficiency. We experimentally evaluate our solutions with real job traces. The results show that the integrated algorithms generally perform better in terms of job response time than employing separate task allocation and scheduling algorithms.

The rest of this paper is organized as follows. Section II describes the related work. Section III introduces the system model and the objective of our problem. Section IV presents our proposed algorithms. The experimental setting and results are discussed in Sections V and VI respectively. Finally, Section VII concludes the paper.

II. Related work

Some recent work has studied distributed job execution. Hung *et al.* [3] studied scheduling strategies for jobs that run across geo-distributed datacenters. Jobs are composed of multiple tasks and each task has a designated datacenter to execute. A job is said to be completed when all the tasks of the job are finished. Hung *et al.* proposed a Workload-Aware Greedy Scheduling (SWAG) strategy which schedules the jobs greedily based on their estimated completion times. It is a state-of-the-art scheduling strategy for optimizing the job response times for distributed job execution. We shall take advantage of this strategy to schedule jobs in our work. Different from [3], we consider a more general scenario in which each task can possibly be executed at multiple locations due to data replication.

Guan *et al.* [4] studied fair resource allocation among jobs requiring distribute execution when their resource demands exceed the resource capacities available at geo-distributed sites. They extended the conventional max-min fairness for

resource allocation to distributed job execution and defined a new resource allocation policy called Aggregated Max-min Fairness which requires the aggregate resource allocation across all sites to be max-min fair. However, the case of data replication across multiple sites was not considered.

Beaumont *et al.* [5] established several theoretical results for scheduling tasks with replicated inputs under simplifying assumptions such as homogeneous machines with the same processing capacities and homogeneous tasks with the same duration. They studied two variants of the problem: one aiming at minimizing the completion time of the job under the constraint that data locality is preserved for all tasks, and the other aiming at minimizing the number of non-local tasks under the constraint that machines are never left idle. Links were established between these variants and semi-matchings in graph theory. However, this work focused on only a single job with independent tasks. In contrast, we consider a system continuously serving jobs that are released over time where scheduling issues arise not only among tasks within a job but also among different jobs.

Chen *et al.* [6] proposed a task assignment strategy to achieve max-min fairness across the jobs in terms of their completion times. They assumed the tasks of jobs could be migrated among datacenters at the cost of network transfer. The data to be processed by each task was assumed to be available at one datacenter only. Besides, an offline setting was considered in which the total number of tasks to execute for all jobs was assumed to be within the total computing capacity of all the datacenters. In contrast, we consider the scenario in which the data to be processed are replicated across multiple locations and focus on the online setting in which jobs may need to be queued when there is not enough computing capacity to process all the outstanding tasks immediately.

Im *et al.* [7] considered the problem of scheduling jobs with heterogeneous demands on multiple servers. Each server has a certain computing capacity and can run multiple jobs simultaneously as long as the total demand of the jobs does not exceed the server's capacity. Approximation algorithms were developed to minimize the total completion time of all jobs by making scheduling decisions according to the job lengths, demands and volumes. The problem was studied in the offline setting and with the requirements that each job must be assigned to one machine and jobs must be scheduled non-preemptively. Different from [7], we focus on distributed job execution in the online setting and allow a job to be preempted by other jobs in the execution. This is natural when each job is composed of small tasks processing independent data partitions.

III. System Model

We consider a distributed system consisting a set of m sites: $S_1, S_2, ..., S_m$. Each site models a cluster or a datacenter. Each site j has a processing capacity u_j, which can be understood as the number of computing slots available at the site.

Each job to execute in the system includes a set of tasks that process different data partitions and can run in parallel. Each task can be executed at one or multiple sites depending on the data availability. These sites are referred to as the *available sites* of the task. Figure 1 shows an example job composed of 3 tasks. The data partition to be processed by task t_1 is replicated at sites S_1, S_2 and S_3. Thus, the available sites of task t_1 include sites S_1, S_2 and S_3. Similarly, the available sites of task t_2 include sites S_2 and S_4, and the available sites of task t_3 include sites S_1 and S_3. To execute a job, we need to assign each task to one site for processing. We refer to the site at which a task runs as the *processing site* of the task.

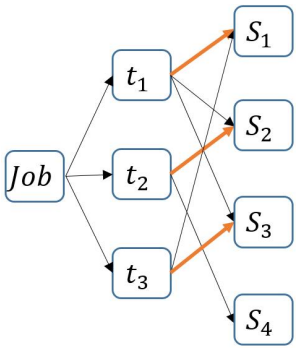


Fig. 1. Example of available sites and processing sites

A job is completed when all of its tasks are finished. The response time of a job is the duration from its release to its completion. Our objective is to schedule the job execution to minimize the average response time of all the jobs.

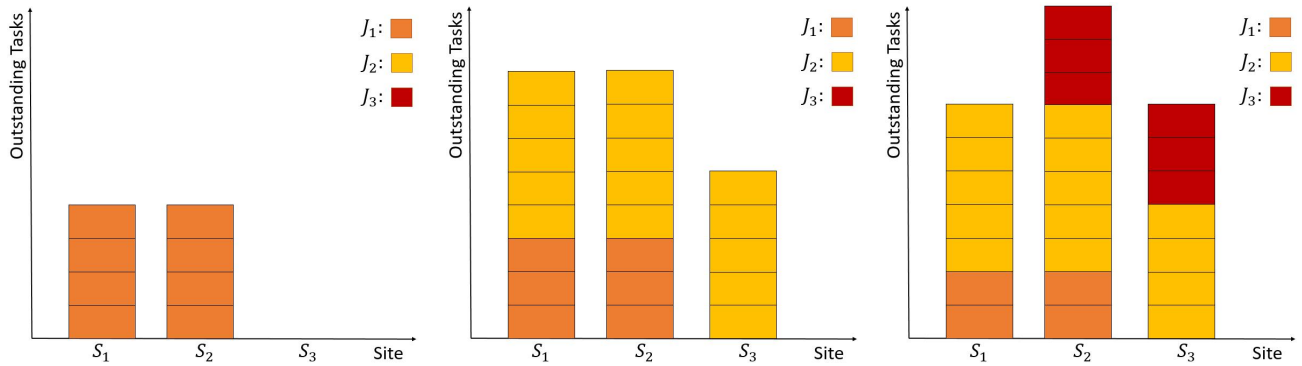
IV. Job Secheduling Algorithms

In this section, we develop scheduling solutions for our problem. To schedule the jobs across multiple sites, we need to determine the processing site for each task of a job and decide the execution order of the tasks at each site. First, we consider addressing these two issues separately. In Section IV-A, we propose an algorithm for choosing the processing site of each task from its available sites. The algorithm can be used together with any scheduling algorithm that decides the execution order of the tasks assigned to each site. Then, we consider addressing the above two issues in an integrated manner. In Sections IV-B and IV-C, we propose two holistic solutions that determine the processing sites and execution order of tasks together.

A. Balanced Task Allocation across Jobs

Since a job is completed only when all of its tasks are finished, the completion time of a job is normally decided by the site that has the largest number of tasks to run normalized by the site capacity. Thus, an intuitive strategy to reduce the job response time is to balance the task allocation among the sites.

A naïve method is to allocate tasks to sites for each job independently. Consider an example of 3 jobs. Table I shows the release time and task number of each job. For simplicity, in this example, we assume that all the tasks of a job have the same set of available sites and the duration of each task is 1



(a) at time t = 0 (s) (b) at time t = 1 (s) (c) at time t = 2 (s)

Fig. 2. Outstanding tasks to execute by balancing task allocation within each job

TABLE I
SETTINGS OF THE EXAMPLE

Job	Release Time (s)	Number of Tasks	Available Sites of Tasks
J_1	0	8	$\{S_1, S_2\}$
J_2	1	15	$\{S_1, S_2, S_3\}$
J_3	2	6	$\{S_1, S_2\}$

second. We also assume that each site has only one computing slot so that tasks are executed one at a time. Figures 2(a), (b) and (c) show the task allocations for jobs J_1, J_2 and J_3 respectively when each job is considered separately. For J_1, since all of its tasks have the same available sites S_1 and S_2, the tasks are uniformly distributed between these two sites (see Figure 2(a)). Similarly, the tasks of J_2 are uniformly distributed among sites S_1, S_2 and S_3. When J_2 arrives, sites S_1 and S_2 would have both finished one task of J_1. Thus, after J_2 arrives, the task numbers to execute at S_1, S_2 and S_3 are 8, 8 and 5 respectively (see Figure 2(b)). For J_3, its tasks are also uniformly distributed between the available sites S_2 and S_3. Suppose the jobs are executed in the order of J_1, J_2 and J_3 at all sites. When J_3 arrives, sites S_1 and S_2 would have both finished two tasks of J_1; and site S_3 would have finished one task of J_2. Thus, after J_3 arrives, the task numbers to execute at S_1, S_2 and S_3 are 7, 10 and 7 respectively (see Figure 2(c)). As a result, J_1, J_2 and J_3 would be completed at time 4, 9 and 12 seconds respectively. So, the average job response time is $((4-0)+(9-1)+(12-2))/3 = 22/3$ seconds.

The above method considers each job independently. It does not consider the workload already allocated to the sites in the task allocation of a new job. Different jobs can have different available sites for their tasks and the distribution of available sites may be skewed. As a result, task allocation accumulated over multiple jobs may be quite unbalanced, which can adversely affect the job completion times. In the example of Figure 2(b), if more tasks of job J_2 are allocated to site S_3 than to sites S_1 and S_2, we can achieve a more balanced overall task allocation between these two sites, thereby improving the response time of J_2. This motivates us to design a task assignment algorithm which looks at not only the available sites of the new job to allocate but also the existing task distribution among the sites. The goal is to balance the overall task allocation.

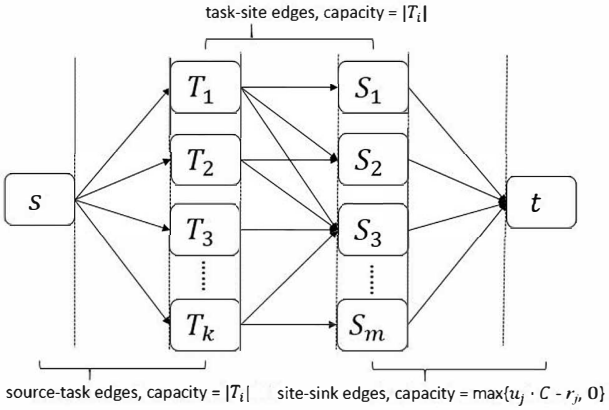


Fig. 3. Flow network

Given the available sites of the tasks in a new job, balanced task allocation can be modeled as a maximum flow problem in a flow network. We first group all the tasks by their available sites. All the tasks sharing the same set of available sites are put into the same group. Suppose that a new job has a total of n tasks which are composed of k task groups: $T_1, T_2, ..., T_k$. We construct a flow network (see Figure 3) with a source node s, a sink node t, and a set of $(k+m)$ nodes, where k is the number of task groups and m is the number of sites. Among the $(k+m)$ nodes, there are k nodes each representing a task group, and m nodes each representing a site. For ease of presentation, we shall use T_i to refer to both a task group and its corresponding node in the flow network, and use S_j to refer to both a site and its corresponding node in the flow network. A set of edges connects the source node to each node of a task group. The capacity of the edge from the source node to node T_i is set to $|T_i|$, i.e., the number of tasks in group T_i. In addition, a set of edges connects each node of a task group to the nodes of its available sites. The capacity of the edge from node T_i to a node S_j of its available sites is also set to

$|T_i|$. Finally, a set of edges connects each node of a site to the sink node. For each site S_j, let r_j denote the number of tasks already allocated to but not yet started at site S_j when the new job arrives. Then, the capacity of the edge from node S_j to the sink node is set to $\max\{u_j \cdot C - r_j, 0\}$, where u_j is the processing capacity of site S_j and C is a constant.

It is easy to verify that there is a one-to-one correspondence between the integral flows in the flow network constructed and the (partial) task allocations of the job satisfying that every site is allocated at most C tasks per computing slot overall unless $r_j > u_j \cdot C$ (the existing tasks allocated already exceed $u_j \cdot C$). In fact, given an integral flow f in the network, we can induce a task allocation of the new job as follows. For each task group T_i of the new job and each of its available sites S_j, we assign $f(T_i, S_j)$ tasks of group T_i to site S_j, where $f(T_i, S_j)$ is the amount of flow going through the edge from node T_i to node S_j in the flow network. In such a task allocation, the overall number of tasks assigned to each site, including the r_j tasks allocated from earlier jobs but not yet started, must be capped at $\max\{u_j \cdot C, r_j\}$ since the edges connecting each node S_j to the sink node have the capacity $\max\{u_j \cdot C - r_j, 0\}$. Vice versa, given a task allocation of the new job such that no site is allocated more than C tasks per computing slot overall unless $r_j > u_j \cdot C$, we can induce a flow in the flow network as follows. For each task group T_i of the new job and each of its available sites S_j, the flow $f(T_i, S_j)$ from node T_i to node S_j is set to the number of tasks in group T_i assigned to site S_j. In addition, the flow $f(s, T_i)$ from the source node to node T_i is set to the total number of tasks allocated in group T_i, and the flow $f(S_j, t)$ from node S_j to the sink node is set to the total number of tasks of the new job assigned to site S_j. Obviously, such a flow meets the capacity constraint, skew symmetry and flow conservation properties.

Since all the edges in the flow network constructed have integral capacities, the maximum flow of the network must have an integral flow value. Thus, by computing the maximum flow of the flow network, we can answer the question of whether there exists a task allocation such that no site is assigned more than C tasks per computing slot overall unless $r_j > u_j \cdot C$. If the flow value of the maximum flow is equal to the total number of tasks $\sum_{i=1}^{k} |T_i| = n$, such a task allocation exists. Otherwise, such a task allocation does not exist. As a result, balanced task allocation can be solved as follows. Since the new job has n tasks in total and there are m sites, there must exist one site that needs to be assigned at least $\lceil \frac{n}{\sum_{j=1}^{m} u_j} \rceil$ tasks per computing slot. On the other hand, to accommodate all the $n + \sum_{j=1}^{m} r_j$ tasks (including the outstanding tasks from earlier jobs), each site S_j needs to run at most a total of $n + r_j$ tasks and hence at most $\lceil \frac{n+r_j}{u_j} \rceil$ tasks per computing slot. Therefore, we can perform a binary search in the range $[\lceil \frac{n}{\sum_{j=1}^{m} u_j} \rceil, \max_{1 \le j \le m} \lceil \frac{n+r_j}{u_j} \rceil]$ to identify the lowest C value such that a task allocation exists to assign no more than C tasks per computing slot to each site S_j. This is the balanced task allocation that minimizes the largest number of tasks per computing slot received by a site. We refer to this algorithm as Balanced Task Allocation across Jobs (BTAaJ). Algorithm 1 shows the details of the BTAaJ algorithm.

Algorithm 1 Balanced Task Allocation across Jobs

Input:
number of tasks in a new job: n;
the available site set of each task in the new job;
number of sites: m;
number of remaining tasks at each site: $\{r_1, r_2, ..., r_m\}$;
Output:
allocation of the new job's tasks to sites;

```
construct the flow network for the new job;
initialize the lower bound of C as C_lower = ⌈n / Σ_{j=1}^{m} u_j⌉;
initialize the upper bound of C as C_upper = max_{1≤j≤m} ⌈(n+r_j)/u_j⌉;
while C_lower < C_upper do
    C = ⌊(C_lower + C_upper)/2⌋,
    set the capacity of each edge (S_j, t) to max{u_j · C − r_j, 0};
    compute the maximum flow f of the network;
    if |f| = n then
        C_upper = C;
    else
        C_lower = C + 1;
    end if
end while
C = C_lower;
set the capacity of each edge (S_j, t) to max{u_j · C − r_j, 0};
compute the maximum flow of the network;
derive the task allocation of the new job from the maximum flow;
```

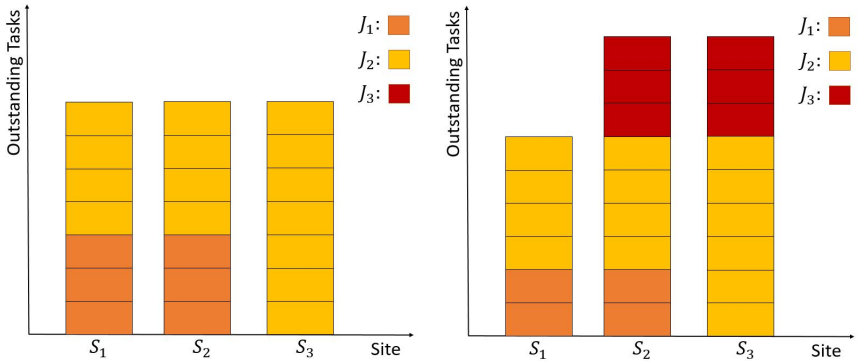
We use the same example of Table I to illustrate BTAaJ. Since all the sites are empty when J_1 arrives, BTAaJ uniformly distributes its tasks between the available sites S_1 and S_2 (see Figure 2(a)). When J_2 arrives, the remaining task numbers at S_1, S_2 and S_3 are 3, 3 and 0. Thus, BTAaJ assigns four tasks to site S_1, four tasks to site S_2 and seven tasks to site S_3 to balance the overall allocation among the sites (see Figure 4(a)). Suppose the jobs are again executed in the order of J_1, J_2 and J_3 at all sites. When J_3 arrives, the remaining task numbers at S_1, S_2 and S_3 are 6, 6 and 6. Thus, BTAaJ uniformly distributes the tasks of J_3 between the available sites S_2 and S_3 (see Figure 4(b)). As a consequence, J_1, J_2 and J_3 would be completed at time 4, 8 and 11 seconds respectively. Therefore, the average job response time is $((4-0) + (8-1) + (11-2))/3 = 20/3$ seconds.

We remark that after the task allocations by the BTAaJ algorithm, any scheduling algorithm can be used to determine the execution order of the tasks at each site.

B. Schedule Conscious Task Allocation

The above task allocation algorithm is oblivious to the job scheduling strategy. There are a wide variety of job scheduling strategies. Different scheduling strategies prioritize jobs in different ways. If the task allocations of the jobs can take into consideration the job priorities by the scheduling strategy, it may be possible to further enhance the quality of the overall solution. Next, we design integrated solutions that combine task allocation and job scheduling strategies.

There has been some studies on the scheduling algorithms to minimize the job completion time for distributed job execu-



(a) at time t = 1 (s) (b) at time t = 2 (s)

Fig. 4. Outstanding tasks to execute by BTAaJ

tion. Hung *et al.* [3] proposed a SWAG algorithm that greedily serves the job that can be completed the fastest and showed that the SWAG algorithm outperforms the classical First Come First Serve (FCFS) and Shortest Remaining Processing Time (SRPT) based approaches. Thus, we adopt the SWAG algorithm as the base of our integrated solutions.

In a nutshell, whenever a new job arrives or an existing job is completed, the SWAG algorithm computes a new order of all the outstanding jobs and all the sites would then follow the order to execute the tasks locally. SWAG prioritizes jobs by using a simple heuristic for estimating their completion times and iteratively adds jobs to the new order one at a time. Specifically, suppose the order of the first h jobs to run has been determined as $J_1, J_2, ..., , J_h$. Let $q_{i,j}$ denote the number of job J_i's remaining tasks to execute at site S_j. Then, for the first h jobs, the accumulated number of tasks to execute at each site S_j is $\sum_{i=1}^{h} q_{i,j}$. Consider another job J that has x_j remaining tasks to execute at each site S_j. If J is scheduled as the $(h+1)$-th job to run, its completion time is estimated as $t_c + \max_{1 \le j \le m} \frac{\sum_{i=1}^{h} q_{i,j} + x_j}{u_j}$ (where t_c is the current time and u_j is the processing capacity of site S_j[1]) since a job is completed when all of its tasks are finished. The SWAG algorithm estimates the completion times of all the jobs yet to be ordered and selects the job with the earliest completion time to append to the job order. Then, SWAG continues to update the completion time estimations of the remaining jobs and pick the next job until all the jobs are ordered.

In our problem, each task of a job can have multiple available sites to execute. This provides the opportunity to optimize the completion time of a new job by adjusting its task allocation. That is, given the available sites of the new job's tasks, we would like to derive the x_j values that minimize the job completion time $t_c + \max_{1 \le j \le m} \frac{\sum_{i=1}^{h} q_{i,j} + x_j}{u_j}$. This problem can again be modeled by the flow network constructed in Section IV-A. Specifically, instead of setting the edge capacity from each node S_j to the sink node at $u_j \cdot C$, we can set the edge capacity to $\max\{u_j \cdot C - \sum_{i=1}^{h} q_{i,j}, 0\}$. As a result, each integral flow in the network corresponds to a task allocation of the new job having a completion time at most $t_c + C$ if appended to the job order. A binary search can be conducted to find the lowest C value to allocate all the tasks of the new job. Similar to BTAaJ, the lower and upper bounds for binary search can be set to $\lceil \frac{n}{\sum_{j=1}^{m} u_j} \rceil$ and $\max_{1 \le j \le m} \lceil \frac{n + \sum_{i=1}^{h} q_{i,j}}{u_j} \rceil$ respectively. Then, the lowest C value would be used as the estimated completion time of the new job for the SWAG algorithm to decide the job order. If the new job has the earliest completion time, it is selected as the $(h+1)$-th job to run. Otherwise, its task allocation and the C value would be recomputed when choosing the next job to run. In this way, the task allocation of the new job can be tailored to its priority in the job order, thereby optimizing its completion time. We refer to this algorithm as Schedule Conscious Task Allocation (SCTA). Algorithm 2 shows the details of the SCTA algorithm.

Again, we illustrate SCTA with the example of Table I. The task allocations of J_1 and J_2 by SCTA are the same as those by BTAaJ. When J_1 arrives, to minimize its estimated completion time, its tasks are uniformly distributed between the available sites S_1 and S_2 (see Figure 2(a)). When J_2 arrives, the remaining task numbers at S_1, S_2 and S_3 are 3, 3 and 0. If J_1 is scheduled to run first, it can be completed at time 4 seconds. If J_2 is scheduled to run first, it can be completed at time 6 seconds (by uniformly allocating its tasks among the three sites). Thus, SCTA schedules J_1 to run before J_2. Then, to minimize J_2's estimated completion time, SCTA assigns four tasks to site S_1, four tasks to site S_2 and seven tasks to site S_3 (see Figure 4(a)). When J_3 arrives, the remaining task numbers at S_1, S_2 and S_3 are 6, 6 and 6. By applying SCTA, J_1 would be scheduled to run first since it can be completed at time 4 seconds which is the earliest. Then, J_3 would be scheduled to run next since it can be completed at time 6 seconds if executed after J_1, while J_2 can only be completed at time 8 seconds if executed after J_1. Finally, J_2 would be scheduled to run last (see Figure 5). As a result, J_1, J_2 and J_3 would be completed at time 4, 12 and 6 seconds respectively. Thus, the average job response time is $((4-0) + (12-1) + (6-2))/3 = 19/3$ seconds.

[1]For the comparison purpose, t_c can be omitted from the computation.

Algorithm 2 Schedule Conscious Task Allocation

Input:
number of sites: m;
outstanding jobs: $J_1, J_2, ..., J_g$;
allocation of the remaining tasks of each outstanding job J_i $(i = 1, 2, ..., g)$: $\{q_{i,1}, q_{i,2}, ..., q_{i,m}\}$;
number of tasks in a new job J_{g+1}: n;
the available site set of each task in the new job J_{g+1};
Output:
allocation of the new job's tasks to sites and the execution order of all jobs;

construct the flow network for the new job J_{g+1};
initialize $\mathcal{Q}$ as an empty job order;
initialize the accumulated task number $r_j = 0$ for each site S_j $(j = 1, 2, ..., m)$;
for each $h = 1$ to $g + 1$ **do**
  if the new job J_{g+1} is not in $\mathcal{Q}$ **then**
    initialize the lower bound of C as $C_{lower} = \lceil \frac{n}{\sum_{j=1}^{m} u_j} \rceil$;
    initialize the upper bound of C as $C_{upper} = \max_{1 \le j \le m} \lceil \frac{n + r_j}{u_j} \rceil$;
    while $C_{lower} < C_{upper}$ **do**
      $C = \lfloor (C_{lower} + C_{upper})/2 \rfloor$,
      set the capacity of each edge (S_j, t) to $\max\{u_j \cdot C - r_j, 0\}$;
      compute the maximum flow f of the network;
      if $|f| = n$ **then**
        $C_{upper} = C$;
      else
        $C_{lower} = C + 1$;
      end if
    end while
    $C = C_{lower}$;
    set the capacity of each edge (S_j, t) to $\max\{u_j \cdot C - r_j, 0\}$;
    compute the maximum flow of the network;
    derive the task allocation of the new job J_{g+1} from the maximum flow: $\{q_{(g+1),1}, q_{(g+1),2}, ..., q_{(g+1),m}\}$;
  end if
  compute the estimated completion time $e_i = \max_{1 \le j \le m}\{r_j + q_{i,j}\}$ for each job J_i not in $\mathcal{Q}$;
  $l = \arg\min_i e_i$;
  append job J_l to $\mathcal{Q}$;
  $r_j = r_j + q_{l,j}$, for each $j \in \{1, 2, ..., m\}$;
end for

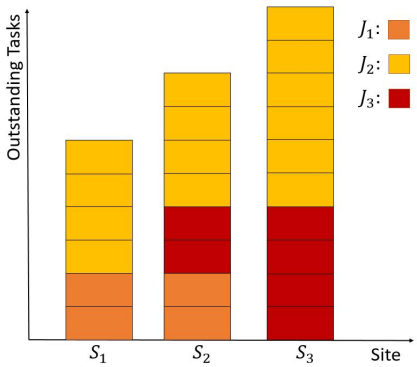


Fig. 5. Outstanding tasks to execute at time t = 2 (s) by SCTA

C. Adaptive Task Allocation

So far, all the solutions we have developed focus on the task allocation of a new job when it arrives. Once determined, the task allocation of the job is fixed and does not change afterwards. This can potentially limit the adaptivity of the task allocation to future job arrivals. In the previous example, when J_3 arrives, the jobs are reordered as J_1, J_3, J_2 (see Figure 5). As a result, due to minimizing J_3's estimated completion time, the overall task allocation among sites S_1, S_2 and S_3 becomes quite unbalanced, which adversely affects the completion time of J_2. In fact, the idea of SCTA to tailor the task allocation of a job to its priority in the job order can be applied to not only a new job but also all the outstanding jobs in the system. To improve the adaptivity, we can reallocate the remaining tasks of a job when needed at any time before the job is completed. In the example of Figure 5, if we reallocate the tasks of J_2 when J_3 arrives, that is, we assign six tasks of J_2 to site S_1, four tasks of J_2 to site S_2 and four tasks of J_2 to site S_3, then we can balance the overall task allocation among the sites (see Figure 6). As a consequence, J_2 can be completed earlier at time 10 seconds. Thus, the average job response time can be reduced to $((4-0)+(10-1)+(6-2))/3 = 17/3$ seconds.

By this motivation, we propose an algorithm called Adaptive Task Allocation (ATA) that is allowed to adjust the task allocation of all the outstanding jobs to optimize the job completion times. Again, we base our ATA algorithm on SWAG which is a state-of-the-art scheduling algorithm for distributed job execution. In computing a new order of the jobs, for each outstanding job, we would like to compute the best task allocation for its remaining tasks that minimizes its estimated completion time. Naturally, this can also be implemented by the flow network transformation and binary search techniques discussed for a new job in Section IV-B. Algorithm 3 shows the details of the ATA algorithm.

The computational complexity of the ATA algorithm can be considerably higher than that of the SCTA algorithm. In the ATA algorithm, to determine a new order of g jobs, we need to compute $O(g^2)$ task allocations since the task allocations of all the jobs not yet ordered are recomputed in each iteration. To improve the computational efficiency, rather than computing the exact best task allocation for each job, we can employ heuristics to derive a good task allocation for each job that minimizes the job completion time in an approximate manner. Next, we propose a simple greedy heuristic that can be used

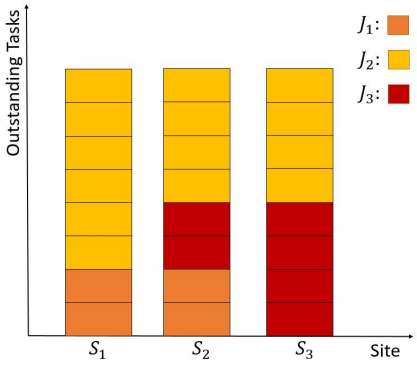


Fig. 6. Outstanding tasks to execute at time t = 2 (s) by ATA

Algorithm 3 Adaptive Task Allocation

Input:
number of sites: m;
outstanding jobs: $J_1, J_2, ..., J_g$;
number of remaining tasks in each outstanding job J_i : n_i;
the available site set of each task in each outstanding job J_i; a new job J_{g+1};
number of tasks in the new job J_{g+1}: n_{g+1};
the available site set of each task in the new job J_{g+1};
Output:
allocation of tasks to sites for all jobs and the execution order of all jobs;

construct the flow network for each job J_i $(i = 1, 2, ..., g+1)$;
initialize $\mathcal{Q}$ as an empty job order;
initialize the accumulated task number $r_j = 0$ for each site S_j $(j = 1, 2, ..., m)$;
for each $h = 1$ to $g+1$ **do**
  for each job J_i not in $\mathcal{Q}$ **do**
    initialize the lower bound of C as $C_{lower} = \lceil \frac{n_i}{\sum_{j=1}^{m} u_j} \rceil$;
    initialize the upper bound C as of $C_{upper} = \max_{1 \le j \le m} \lceil \frac{n_i + r_j}{u_j} \rceil$;
    while $C_{lower} < C_{upper}$ **do**
      $C = \lfloor (C_{lower} + C_{upper})/2 \rfloor$,
      set the capacity of each edge (S_j, t) in the flow network for J_i to $\max\{u_j \cdot C - r_j, 0\}$;
      compute the maximum flow f of the network for J_i;
      if $|f| = n_i$ **then**
        $C_{upper} = C$;
      else
        $C_{lower} = C + 1$;
      end if
    end while
    $C = C_{lower}$;
    set the capacity of each edge (S_j, t) in the flow network for J_i to $\max\{u_j \cdot C - r_j, 0\}$;
    compute the maximum flow f of the network for J_i;
    derive the task allocation of job J_i from the maximum flow: $\{q_{i,1}, q_{i,2}, ..., q_{i,m}\}$;
    compute the estimated completion time $e_i = \max_{1 \le j \le m}\{r_j + q_{i,j}\}$ for job J_i;
  end for
  $l = \arg\min_i e_i$;
  append job J_l to $\mathcal{Q}$;
  $r_j = r_j + q_{l,j}$, for each $j \in \{1, 2, ..., m\}$;
end for

as a substitute for the flow network transformation and binary search techniques in the ATA algorithm.

The main idea of the greedy heuristic is to allocate the tasks of a job sequentially in a water-filling manner (see Figure 7). Suppose that the remaining tasks of a job are composed of k task groups $T_1, T_2, ..., T_k$, where the tasks in the same group share the same set of available sites. The greedy heuristic allocates the tasks one group at a time in descending order of group size. When allocating a task group T_i, it considers all the available sites of the group. Let $S_{j_1}, S_{j_2}, ...S_{j_l}$ denote the available sites, and let $r_{j_1}, r_{j_2}, ..., r_{j_l}$ denote the numbers of tasks currently allocated to these sites. Without loss of generality, assume that the available sites are sorted in ascending order of current workload, i.e., $r_{j_1} \le r_{j_2} \le ... \le r_{j_l}$. Define $r_{j_{l+1}} = \infty$. Let $|T_i|$ denote the number of tasks

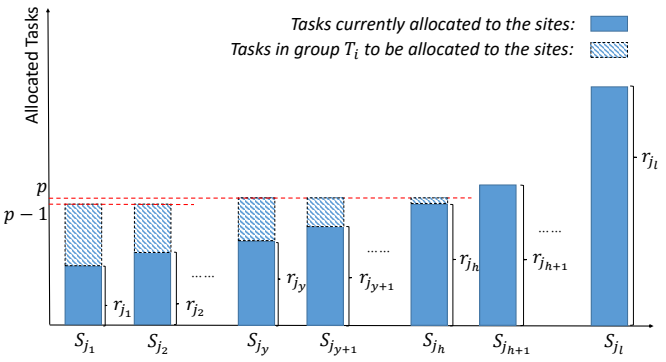


Fig. 7. Task Assignment for ATA-Greedy

in the task group T_i. To decide where to allocate the task group T_i, the greedy heuristic finds the smallest index h satisfying $\sum_{x=1}^{h-1}(r_{j_h} - r_{j_x}) < |T_i| \le \sum_{x=1}^{h}(r_{j_{h+1}} - r_{j_x})$. Then, the tasks in group T_i would be allocated to sites $S_{j_1}, S_{j_2}, ..., S_{j_h}$. Let p (where $r_{j_h} < p \le r_{j_{h+1}}$) be the integer satisfying $\sum_{x=1}^{h}((p-1) - r_{j_x}) < |T_i| < \sum_{x=1}^{h}(p - r_{j_x})$ and let $y = \sum_{x=1}^{h}(p - r_{j_x}) - |T_i|$. The greedy heuristic allocates $(p-1) - r_{j_x}$ tasks to each site S_{j_x} where $1 \le x \le y$, and allocates $p - r_{j_x}$ tasks to each site S_{j_x} where $y < x \le h$. As a result, the total number of tasks allocated is $\sum_{x=1}^{y}((p-1) - r_{j_x}) + \sum_{x=y+1}^{h}(p - r_{j_x}) = \sum_{x=1}^{h}(p - r_{j_x}) - y = |T_i|$. In this way, the accumulated numbers of tasks allocated to the available sites are balanced as much as possible. We refer to the ATA algorithm instantiated with the greedy heuristic as ATA-Greedy.

D. Discussion

In our proposed solutions, we simply use the number of remaining tasks to execute as an estimate of the job response times. We do not make use of any knowledge on task durations. This is because it may not be easy to make good predictions on the task durations for all the applications [8]. Not relying on task duration predictions will allow our solutions to be applicable to a wider range of scenarios. In the next section, we shall conduct experimental evaluations using job traces with realistic task durations and show that our solutions are effective in optimizing the average job response time. We remark that if decent estimates on the task durations are available when jobs are released, our solutions can be further enhanced by incorporating such information in the task assignment and scheduling strategies.

In our discussion, for simplicity, we have assumed that each job includes a single stage in which all the tasks can run in parallel. If a job has multiple stages with dependency constraints, the tasks of the first stage normally process raw data inputs, whereas the tasks of subsequent stages aggregate the outputs of the first-stage tasks and are often executed at one site [3]. Thus, in this case, our solutions can primarily be used to assign and schedule tasks in the first stage.

V. Experimental Setup

We conduct extensive simulations to compare various scheduling solutions. This section describes the simulation settings.

Job traces: We use two realistic job traces to drive the simulations: a Facebook trace and a Google trace. The Facebook trace is the trace FB-2010_samples_24_times_1hr_0.csv from the SWIM workload repository [9], [10], which is generated based on historical workload traces on a 3000-machine cluster at Facebook. The trace contains 24024 jobs and specifies the amount of data processed by each job. We derive the number of tasks in each job by assuming that there is one task per 1 GB data to process. As a result, there are a total of 1102281 tasks in these jobs. We generate the task durations according to a Pareto distribution with parameter $\beta = 1.259$ [8] and a mean of 2 seconds. The Google trace was collected on a cluster of about 12000 machines for one month at Google [11], [12]. We extract a segment of the trace containing 2944 jobs in a 60-min window. These jobs include 48504 tasks. We derive the task durations from the timestamps of task events recorded in the trace. The task durations show a heavy-tailed distribution and have a mean of 1374.7 seconds. In both traces, each task of a job is assumed to require one computing slot to execute. We scale the inter-arrival times of the jobs in the traces to simulate different levels of system utilization from 40% to 70%.

Site capacity: The default number of sites is set at 30. The sites are denoted as S_1, S_2, ..., S_{30}. The resource capacity of each site is set at 20 computing slots.

Available sites: We assume that for each job, the data inputs to the tasks are distributed among the sites according to a Zipf distribution. Specifically, for each job, we randomly generate a permutation of all the sites. Then, each task of the job is associated with the i-th site in the permutation with a probability proportional to $\frac{1}{i^\alpha}$, where α is the Zipf skew parameter. The higher the value of α, the more skewed the task distribution. To simulate different levels of skewness, we vary α from 0 to 2. When α is set to 0, the expected task distribution is uniform. If the associated site of a task is S_j, then S_j and $(k-1)$ additional sites $S_{j+1}, S_{j+2}, ..., S_{j+k-1}$ are appointed as the available sites of the task. We vary the number of available sites from 1 to 5.

Scheduling methods: We implement all the scheduling solutions described in Section IV. For the BTAaJ algorithm, after determining the task allocation for each new job on its arrival, the jobs are ordered by the SWAG algorithm for execution. The SCTA, ATA and ATA-Greedy algorithms are integrated solutions that combine task allocation and job scheduling (SWAG) strategies. In addition, we also implement the original SWAG algorithm as a baseline for comparison. The original SWAG algorithm does not take advantage of multiple available sites of tasks. It simply allocates each task to its associated site and orders the jobs based on their estimated completion times for execution. Note that none of our scheduling algorithms and SWAG uses the information of task durations since such knowledge is often hard to obtain before execution. The algorithms simply estimate the job completion times by the number of outstanding tasks.

Performance metrics: We study the average response time of all the jobs. The response time of a job is the duration from its arrival to the time when all of its tasks are finished.

VI. Experimental Results

We compare the algorithms by varying the Zipf skew parameter from 0 to 2, and the system utilization from 40% to 70%. Figures 8 and 9 show the results of Google and Facebook traces respectively with each task having 3 available sites. In general, the average job response time increases with the skewness of task distribution for all the algorithms. This is because when the available sites of tasks have a more skewed distribution, it is more difficult to balance the task allocation among the sites, thereby making the job response times longer.

The original SWAG algorithm allocates each task to a fixed available site and does not make use of other available sites. Thus, as can be seen from Figures 8 and 9, it normally has the most skewed task allocation and results in much higher job response times than all our proposed algorithms. The SCTA and ATA algorithms outperform the BTAaJ algorithm. This indicates that combining task allocation and job scheduling strategies are more effective for optimizing job response times than conducting task allocation and job scheduling separately. By reallocating the tasks of outstanding jobs when a new job arrives, ATA further reduces the job response times compared to SCTA. This shows the importance of adjusting task allocations according to future job arrivals. The performance improvement of ATA over SCTA generally increases with the Zipf skew parameter. This demonstrates that ATA is more capable of dealing with the skewness in the distribution of tasks' available sites. ATA-Greedy, the heuristic version of ATA which compromises the quality of task allocation, performs a little worse than ATA but still significantly better than the other algorithms. This implies that the heuristic to greedily allocate tasks to sites with the least loads is a close approximation of the optimal task allocation. Figures 10 and 11 show the cumulative distribution of job response times for various algorithms when the Zipf skew parameter is set at 1. It can be seen that ATA and ATA-Greedy improve the job response time at almost all percentiles. The above performance trends are consistently observed across different levels of system utilization.

Figures 12 and 13 show the average job response times for different numbers of available sites for each task when the Zipf skew parameter is set at 1. In general, a larger number of available sites provide more flexibility in task allocation. Thus, the job response times of our algorithms usually decrease as the number of available sites increases. When each task has only one available site, the job response times of all the proposed algorithms are the same because the task allocations are fixed and all the jobs are executed in the same order decided by SWAG. When each task has more than one available site, the tasks of each job can be distributed to balance the task allocation among the sites and reduce the job

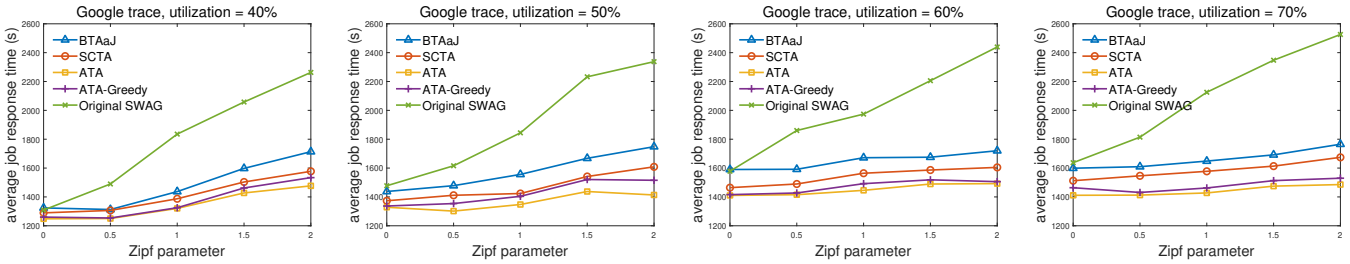


Fig. 8. Average job response time for Google trace (3 available sites for each task)

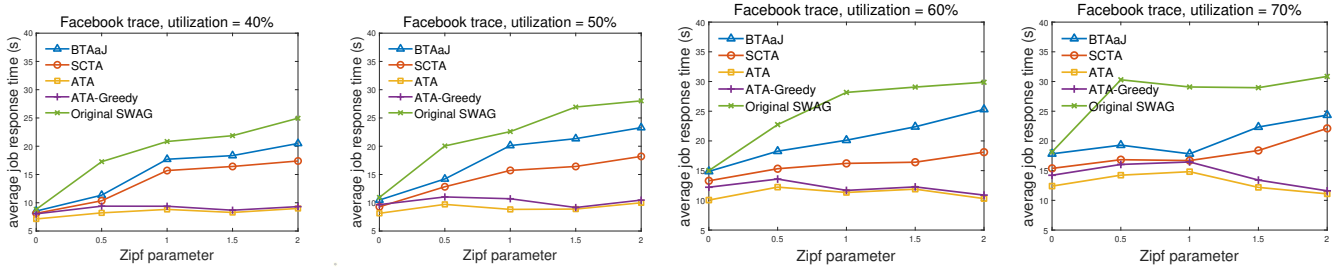


Fig. 9. Average job response time for Facebook trace (3 available sites for each task)

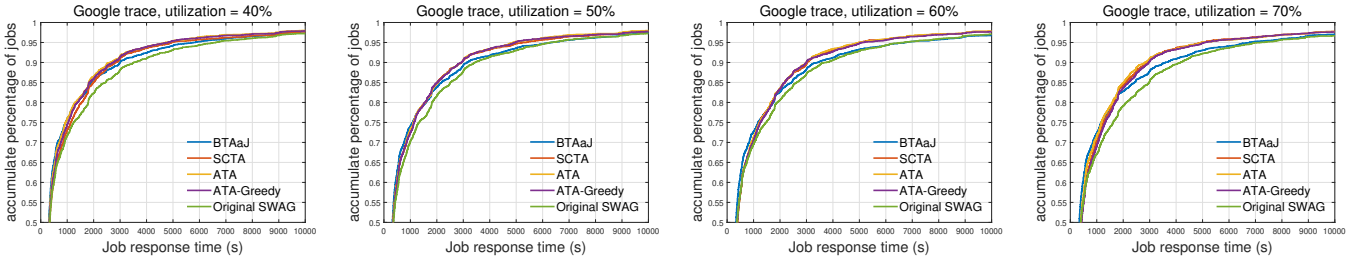


Fig. 10. Cumulative distribution of job response time for Google trace (Zipf parameter = 1, 3 available sites for each task)

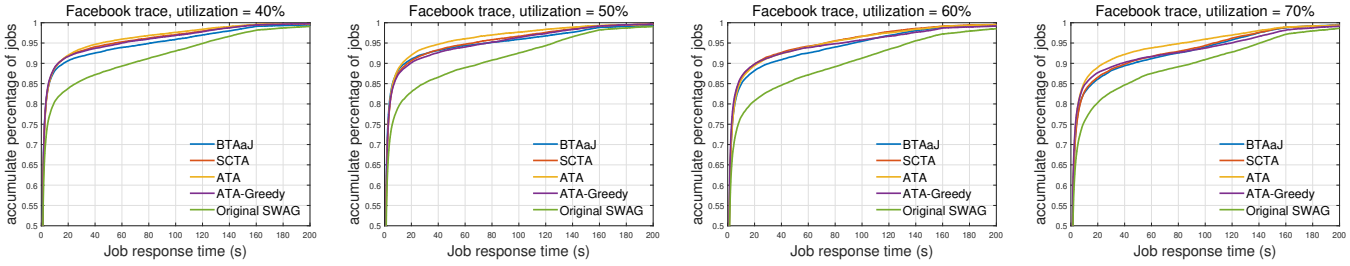


Fig. 11. Cumulative distribution of job response time for Facebook trace (Zipf parameter = 1, 3 available sites for each task)

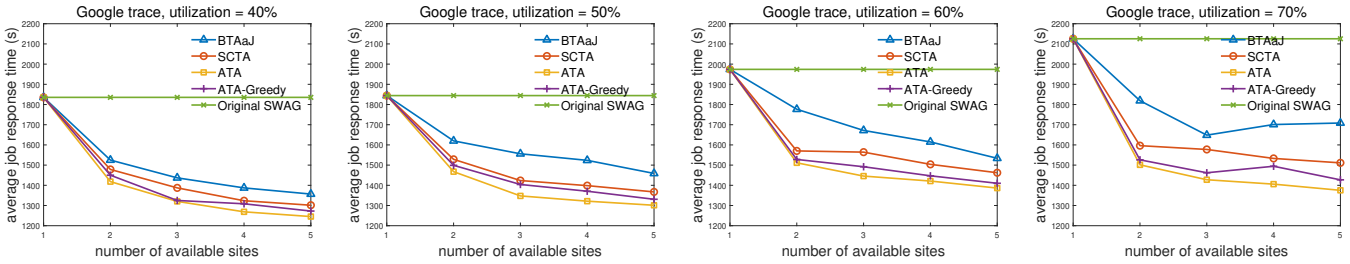


Fig. 12. Average job response time for Google trace (Zipf parameter = 1)

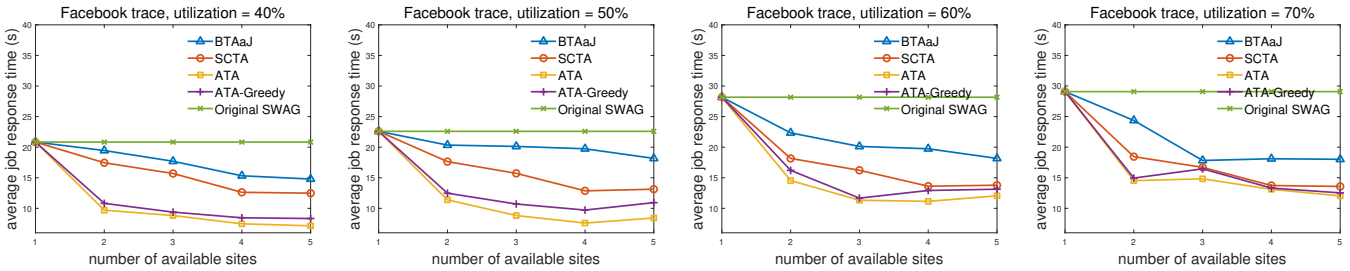


Fig. 13. Average job response time for Facebook trace (Zipf parameter = 1)

response times. The relative performance of the algorithms remains largely unchanged for different numbers of available sites. The performance trends are similar for the results of other Zipf skew parameter values, which are not shown here due to space limitations.

VII. Conclusion

In this paper, we have studied task assignment and scheduling for distributed job execution in which each task of a job may be executed at a subset of all the sites. We model the task assignment as a flow network, and design algorithms to find the balanced task allocation among the sites by solving a maximum flow problem. We further propose a number of integrated solutions to carry out task assignment and job scheduling together. Experiments with real job traces show that these solutions perform significantly better in terms of job response time than conducting task assignment and job scheduling separately and a baseline that allocates each task to a fixed available site.

Acknowledgments

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award MOE-T2EP20121-0005).

References

[1] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 2015, pp. 421–434.

[2] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015, pp. 323–336.

[3] C.-C. Hung, L. Golubchik, and M. Yu, "Scheduling jobs across geo-distributed datacenters," in *Proceedings of the 6th ACM Symposium on Cloud Computing*, 2015, pp. 111–124.

[4] Y. Guan, C. Li, and X. Tang, "On max-min fair resource allocation for distributed job execution," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[5] O. Beaumon, T. Lambert, L. Marchal, and B. Thomasa, "Performance analysis and optimality results for data-locality aware tasks scheduling with replicated inputs," *Future Generation Computer Systems*, vol. 111, pp. 582–598, 2020.

[6] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling jobs across geo-distributed datacenters with max-min fairness," *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, pp. 488–500, 2019.

[7] S. Im, M. Naghshnejad, and M. Singhal, "Scheduling jobs with non-uniform demands on multiple servers without interruption," in *Proceedings of the 35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1–9.

[8] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "GRASS: trimming stragglers in approximation analytics," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014, pp. 289–302.

[9] "Swim's facebook workload suite," https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

[10] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *Proceedings of the 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2011, pp. 390–399.

[11] "Google cluster workload traces," https://github.com/google/cluster-data.

[12] 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 3rd ACM Symposium on Cloud Computing*, 2012, pp. 1–13.