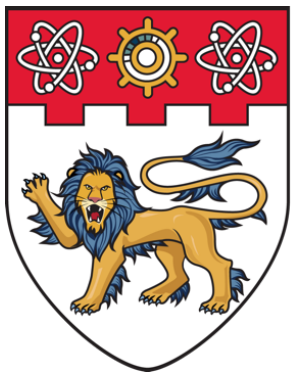


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

TASK ALLOCATION AND SCHEDULING FOR DISTRIBUTED JOB EXECUTION

GUAN Yitong

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2022

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

Date:
1 June, 2022

.....
GUAN Yitong

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

Date:
1 June, 2022

.....
TANG Xueyan

Authorship Attribution Statement

This thesis contains material from 2 papers published in the following peer-reviewed journal(s) / from papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as Yitong Guan and Xueyan Tang. On Task Assignment and Scheduling for Distributed Job Execution. Proceedings of the 22nd IEEE/ACM International Symposium on Clusters, Cloud and Internet Computing (CCGrid), pp. 726-735, 2022.

The contributions of co-authors are as follows:

- Prof. Tang provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript draft.
- I co-designed algorithms.
- All the experimental simulations were completed by me.

Chapter 4 is published as Yitong Guan, Chuanyou Li and Xueyan Tang. On Max-min Fair Resource Allocation for Distributed Job Execution. Proceedings of the 48th International Conference on Parallel Processing. article no. 55, pp. 1-10, 2019.

The contributions of co-authors are as follows:

- Prof. Tang provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript draft.
- I co-designed the policy and algorithms.
- All the experimental simulations were completed by me.

- Dr Li assisted in proving the algorithms in the paper.

Date:
1 June, 2022

.....
GUAN Yitong

Acknowledgments

Foremost, my sincere gratitude goes to my supervisor **Prof. Tang Xueyan**, who has given me continuous support and guidance over the past years in my study and research. His immense knowledge and rich experience, together with his enthusiasm and patience, always motivated me when I met any challenges or difficulties throughout this thesis writing. He is not only a great supervisor but also a respectful mentor for me in my Ph.D study and research life. Also, I would like to express my sincere thanks to **Dr. Li Chuanyou**, for the stimulating discussions and encouragement. Last but definitely not the least, much gratitude and love to my family, who have always been there supporting me materially and spiritually, even though we are far away from each other with limited opportunities to meet.

Contents

List of Figures

Abstract

Data analytic jobs usually require large volumes of data inputs that are available at geographically distributed locations. Gathering all the data at a central location for processing would not only place heavy traffic burdens on the underlying networks but also slow down the job execution. To achieve better performance, it is often preferable to take advantage of data locality by distributing job execution across multiple sites located close to the data to be processed. Efficiency and fairness are two important considerations for sharing resources among concurrently running distributed jobs. In this thesis, we focus on the efficiency and fairness aspects of resource allocation in distributed job execution and study three specific problems, including task assignment and scheduling for improving efficiency of distributed job execution, max-min fair resource allocation for distributed job execution, and generalized max-min fair resource allocation for jobs with tasks executable at multiple sites.

First, we study a task assignment and scheduling problem for improving efficiency of distributed job execution in which the data inputs to the 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 task assignment algorithms 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.

Second, we study fair resource allocation among jobs requiring distributed execution. We extend conventional max-min fairness for resource allocation in a single machine

or machine cluster to distributed job execution over multiple sites and define Aggregate Max-min Fairness (*AMF*) which requires the aggregate resource allocation across all sites to be max-min fair. We show that *AMF* satisfies the properties of Pareto efficiency, envy-freeness and strategy-proofness, but it does not necessarily satisfy the sharing incentive property. We propose an enhanced version of *AMF* to guarantee the sharing incentive property. We present algorithms to compute *AMF* allocations and propose an add-on to optimize the job response times under *AMF*. Experimental results show that compared with a baseline which simply requires the resource allocation at each site to be max-min fair, *AMF* performs significantly better in balancing resource allocation and in average job response time, particularly when the workload distribution of jobs among sites is highly skewed.

When we design the *AMF* policy, we assume that a job contains a set of tasks and each task can be allocated to only one fixed site for data locality. For reliability or fault tolerance considerations, it is not uncommon for the data to be replicated across several sites. As a result, there are multiple choices for placing tasks that preserve data locality. Finally, we generalize *AMF* to such scenarios and define Generalized Aggregate Max-min Fairness (*GAMF*) which also requires the aggregate resource allocation across all sites to be max-min fair. We show that *GAMF* satisfies the properties of Pareto efficiency, envy-freeness and strategy-proofness, but it does not necessarily satisfy the sharing incentive property either. We further enhance *GAMF* to guarantee the sharing incentive property. Experimental results show that *GAMF* outperforms a baseline that conducts max-min fair allocation at each site separately after distributing tasks with the same available sites of data inputs uniformly among these sites.

Chapter 1

Introduction

Big data analytics have been widely used nowadays. Data analytic jobs usually consist of many tasks that process different partitions of data inputs and can run in parallel. With increasing geo-distributed deployments of datacenters by major cloud providers, it has become quite common for massive amounts of data to be generated and stored at geographically distributed locations. Gathering all the data at a central location for processing would not only place heavy traffic burdens on the underlying networks but also slow down the job execution. To achieve better performance, it is often preferable to take advantage of data locality by distributing job execution across multiple sites located close to the data to be processed [51, 61].

Efficiency and fairness are two important considerations for sharing resources among concurrently running distributed jobs. On the one hand, when all the jobs are from one user, efficiency is the most important consideration from the user's perspective – it is desirable to minimize the average response time of all the jobs. On the other hand, when the jobs are from different users that share the computing resources, fairness is an important consideration from the system's perspective – it is preferable to allocate resources among the jobs in a fair manner when there are not enough resources to fully meet the demands of all the jobs.

Here we use an example to illustrate the above issues. Suppose three jobs are running concurrently at three sites. Each job contains 10 tasks, and each site has a capacity that can process 3 tasks at the same time. All the three jobs arrive at time 0. Suppose that each task of the three jobs can be assigned to any of the three sites for data locality and it takes the one unit of time to execute each task. Figure 1.1 shows one possible way

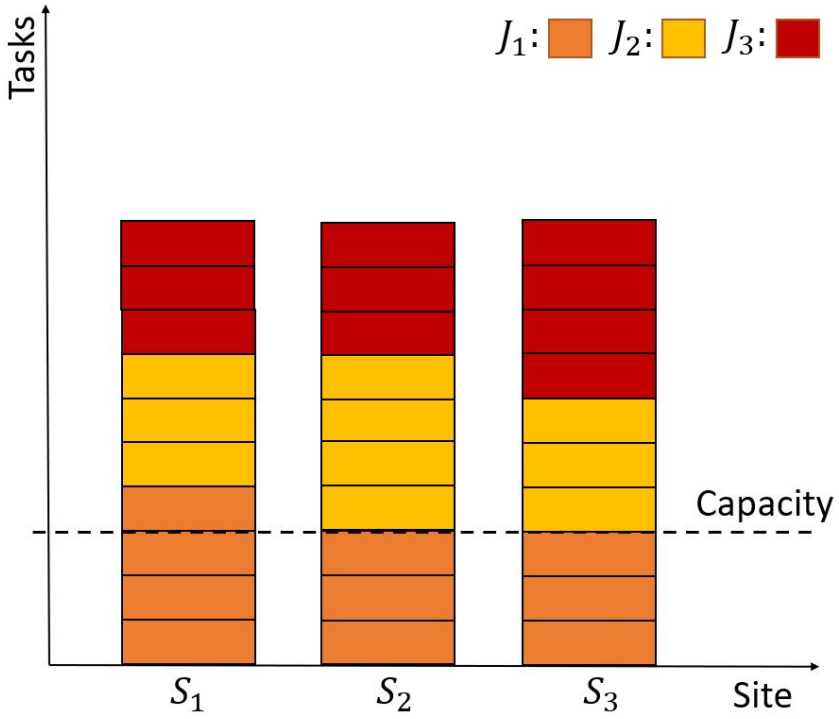


Figure 1.1: An example of three jobs and three sites

of task allocation and scheduling. The tasks of each job are distributed among all the three sites. At each site, the tasks are processed in the order of $J_1 \rightarrow J_2 \rightarrow J_3$. Then in the first three time units, only the tasks of J_1 are executed at all the sites, and this may be unfair to J_2 and J_3. We can see from Figure 1.1 that J_1 would be completed at time 4, J_2 would be completed at time 7, and J_3 would be completed at time 10. Thus, the average job response time is $\frac{4+7+10}{3} = 7$. Figure 1.2 shows another possible way of task allocation and scheduling. The tasks of each job are assigned to one site, and all the jobs would be completed at the same time. This method ensures the fairness of resource allocation among jobs at any time during the period of processing. However, the average job response time increases to $\frac{10+10+10}{3} = 10$. We can see that the task allocation and scheduling make difference to the fairness and efficiency of job execution.

The following questions arise naturally based on the above example:

1) How to define the policy of fairness for allocating resources to the tasks of each job to make sure that the resource allocations among jobs are fair?

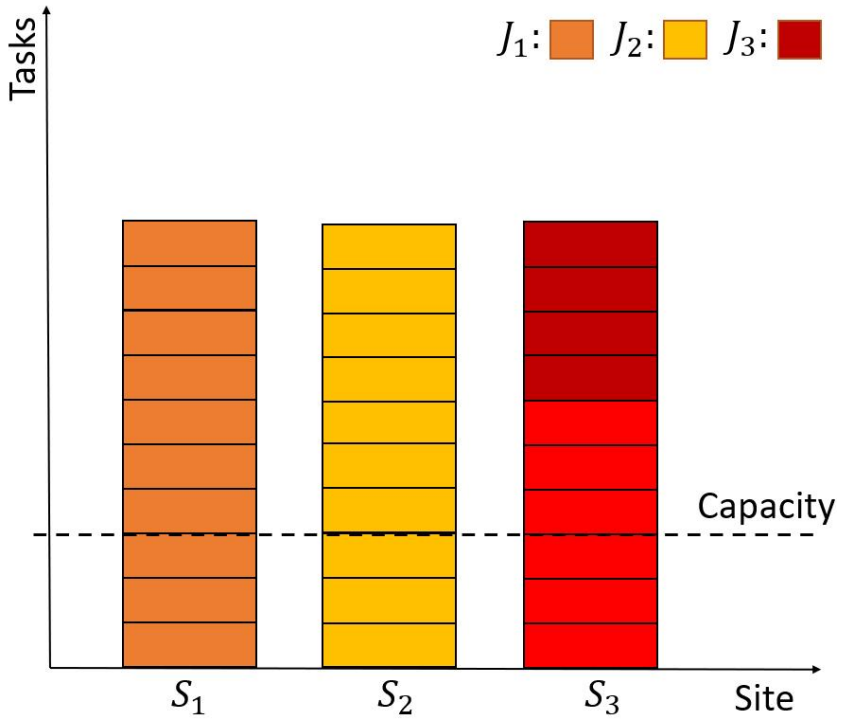


Figure 1.2: An example of three jobs and three sites

2) How to schedule the task execution for jobs to optimize system efficiency in terms of job response times?

3) Suppose each task can be assigned to more than one site, how to choose a site for each task to execute and allocate resources to the tasks of each job to ensure the resource allocations among jobs are fair?

4) Suppose each task can be assigned to more than one site, how to choose a site for each task to execute and schedule the task execution for jobs to improve the efficiency of processing?

We shall answer these questions in this thesis.

1.1 Research Scope

In this thesis, we focus on the efficiency and fairness aspects of resource allocation in distributed job execution and study three specific problems: *task assignment and scheduling*

for improving efficiency of distributed job execution, *max-min fair resource allocation for distributed job execution*, and *generalized max-min fair resource allocation for jobs with tasks executable at multiple sites*.

The first part of the thesis is concerned with the task assignment and scheduling for improving efficiency of distributed job execution introduced as follows. In this part, we formulate and study a task assignment and 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 jobs are released over time and assignment as well as 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 problem. In the first type of solutions, 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 potentially optimize job response times. On completing the task assignment, various scheduling algorithms can be used to determine the task execution order at each location. In the second type of solutions, we design algorithms to assign the 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 average job response time than employing separate task allocation and scheduling algorithms.

The second part of the thesis is concerned with the max-min fair resource allocation for distributed job execution introduced as follows. Max-min fairness is a commonly used definition of fairness for allocating shared resources in computer systems. It focuses on balancing instantaneous resource allocations among different users or jobs. In this part, we extend conventional max-min fairness for resource allocation to distributed job execution over multiple sites (machine clusters or datacenters). We propose a resource allocation policy named Aggregate Max-min Fairness (AMF) which requires the aggregate

resource allocation across all the sites to be max-min fair. We prove that *AMF* satisfies three desirable properties: Pareto Efficiency, envy-freeness and strategy-proofness. However, it does not necessarily satisfy another important property known as sharing incentive. Thus, we enhance *AMF* to guarantee the sharing incentive property. We propose resource allocation algorithms to achieve *AMF* and a heuristic technique to optimize *AMF* allocations for better job response times. We experimentally evaluate *AMF* with real job traces and compare it with a baseline policy which simply requires the resource allocation at each site to be max-min fair. The results show that *AMF* can perform considerably better than the baseline in balancing resource allocation and in average job response time.

The third part of the thesis is concerned with extending the Aggregate Max-min Fairness policy to a more general scenario introduced as follows. In this part, we formulate and study a resource allocation problem for distributed job execution where the data to be processed by each task of the job is possibly available at multiple sites. We generalize *AMF* and propose a resource allocation policy named Generalized Aggregated Max-min Fairness (*GAMF*) where tasks that have the same available sites of data inputs compose a task group of a job. It also requires the aggregate resource allocation across all the sites to be max-min fair. The difference between *AMF* and *GAMF* is that we map the demand of each task of each job to a single site for *AMF* and map the demand of each task group of each job to multiple sites for *GAMF*. We also prove that *GAMF* satisfies three desirable properties: Pareto Efficiency, envy-freeness and strategy-proofness. We enhance *GAMF* to guarantee another important property of sharing incentive. We experimentally evaluate *GAMF* with real job traces and compare it with a baseline policy which first assigns the tasks in a task group to a particular available site of their data inputs and then applies the *AMF* policy. The results show that *GAMF* outperforms the baseline in balancing resource allocation and optimizing average job response time.

1.2 Thesis Contributions

We now give an outline of the thesis and its main results.

In Chapter 2, we review the related work on task allocation and scheduling.

In Chapter 3, we study task assignment and scheduling for improving efficiency of 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 integrated solutions perform significantly better in terms of average job response time than conducting task assignment and job scheduling separately as well as a baseline that allocates each task to a fixed available site of data input.

In Chapter 4, we study fair resource allocation policies for distributed job execution. We propose an Aggregate Max-min Fairness (*AMF*) policy and prove that it satisfies widely recognized properties of Pareto efficiency, envy-freeness and strategy-proofness for fair resource allocation, but the vanilla version of *AMF* does not satisfy the sharing incentive property. We propose an enhanced version of *AMF* to guarantee the sharing incentive property. We present algorithms to implement *AMF* as well as to optimize the job response times under *AMF*. Experiments using real job traces show that *AMF* can significantly reduce unbalanced resource allocation and improve the average job response time compared to a baseline policy which conducts max-min fair allocation at each site separately, and the improvements are more substantial when the workload distribution of jobs among sites is more skewed.

In Chapter 5, we generalize the *AMF* policy in Chapter 4 to address the availability of input data to jobs at multiple sites. We propose a *GAMF* policy which also satisfies Pareto efficiency, envy-freeness and strategy-proofness for fair resource allocation. We further propose an enhanced version of *GAMF* to guarantee the sharing incentive property. We present heuristics to optimize *GAMF* in terms of job response times. We use real job traces to evaluate our algorithms and find that *GAMF* performs much better than a baseline that conducts max-min fair allocation using *AMF* after assigning tasks with the same available sites of data inputs to a particular available site.

In Chapter 6, we conclude the thesis and outline some future research directions.

Chapter 2

Literature Review

In this chapter, we summarize some related work, which addresses the topics of job scheduling and resource allocation in distributed systems. In this thesis, we focus on two major aspects of resource allocation in distributed job execution: fairness and efficiency. Thus, we review some literature about fairness and efficiency separately.

2.1 Fairness

In modern data intensive computing, when there are not enough resources to fully meet the demands of all the jobs, it becomes critical to allocate resources fairly among the jobs. In the previous works, a large number of mechanisms have been proposed for fair resource allocation. In this section, we organize the literatures related to fairness into three sections as follows.

2.1.1 Max-min Fairness and Applications

Max-min fairness is a popular policy for allocating shared resources among competing demands [12]. If an allocation achieves max-min fairness, the allocation of any participant cannot be increased unless the allocation of some other participant having an equal or smaller allocation is decreased. Radunovic and Le Boudec [52] presented a unified framework of max-min fairness, which covers all the past results. They proved that max-min fairness is always achievable in compact and convex sets of allocations and gave a general purpose algorithm to compute the max-min fair allocation whenever it exists.

Many resource allocation algorithms have been proposed based on max-min fairness for various applications. Chen *et al.* [18, 19] proposed a task assignment strategy that is designed to achieve max-min fairness (essentially lexicographical minimization) across the jobs in terms of their completion times. They assumed that the data to be processed by each task of a job is available at one datacenter only and the tasks of jobs could be migrated among datacenters at the cost of network transfer. 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, so there is not really any resource contention. In contrast, in this thesis, we consider the scenario in which the data to be processed are possibly replicated across multiple locations and focus on fair resource allocation in 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. It is also worth pointing out that due to heterogeneous job sizes, max-min fairness in job completion times can be very different from max-min fairness in instantaneous resource allocation. In general, it is not necessary to strive for equal completion times of jobs when they have substantially different sizes.

Angelo *et al.* [21] considered a basic resource allocation problem, where a set of users share a fixed amount of resources. They modeled resource allocation as an optimization problem for minimizing the value of an objective function and established a direct link between the resource allocation problem and the notion of max-min fairness. Specifically, they provided a sufficient condition on the objective function to ensure that the optimal solution is max-min fair. They used a water-filling algorithm to solve the optimization problem under this condition. They also considered an application of max-min fairness in the cellular network.

Sarkar *et al.* [56] studied the fairness in allocating bandwidth for multicast applications. Real-time applications like teleconferencing, audio and video broadcasting transmit data from a sender to a group of receivers. User experiences are strongly interlinked with the bandwidth of the network. Therefore, layered encoding techniques are often used for transmitting the real-time data. Receivers add and drop layers to adapt to the bandwidth congestion. The more layers a user receives, the better the quality of reception is. The work of [56] used max-min fairness as the policy of bandwidth allocation among all

the receivers under resource limitation. It was assumed that the requirements of all the receivers are the same since they want to receive the same number of layers which include the encoded data. In contrast, our work considers resource allocation among jobs that have diverse resource demands.

Max-min fairness is also widely used in many other scenarios of networking. Cao *et al.* [15] proposed a bandwidth allocation scheme among applications, which is equivalent to max-min fair allocation when the utilities of all the applications are equal. Ellen *et al.* [34] proved that round-robin scheduling with windows could be used to achieve max-min throughput fairness in data networks. Huang *et al.* [39] proposed a set of algorithms that allow each network node to determine its max-min fair share in wireless ad-hoc networks. Rubenstein *et al.* [55] identified four desirable fairness properties for multicast networks and defined multicast max-min fairness for multirate sessions.

2.1.2 Extensions of Max-min Fairness

Max-min fairness has been extended along several directions in recent years. Ghodsi *et al.* [30] proposed Constrained Max-Min Fairness to support job placement constraints due to special hardware or software requirements. They modeled the placement constraints of a job by Boolean indicators specifying whether the job can run on each machine. If a job can run on multiple machines, the resources obtained from any of these machines are assumed identical and can be used to process any part of the job. In contrast, in this thesis, we consider distributed job execution in which a job has multiple parallel tasks, each to be executed in one or more designated machine clusters. Thus, the resources obtained from a set of particular machine clusters can only be used to process the tasks assigned to the clusters. Hence, Constrained Max-Min Fairness is not directly applicable to distributed job execution.

Some works considered minimum resource allocation requirements in max-min fairness. Hou *et al.* [37] generalized the classical max-min rate allocation policy with the support of minimum rate requirement and peak rate constraint for each connection. Ros *et al.* [54] considered the problem of allocating max-min rates with minimum rate constraints for connection-oriented networks.

Some other works considered the concept of weight in max-min fairness, where each user is supposed to be assigned a share of resources proportional to its weight. For example, Marbach [48] studied a priority service where users are free to choose the priority of their traffic and charged accordingly by the network. They showed that there exists a unique equilibrium for this game and the bandwidth allocation is weighted max-min fair. Tassiulas *et al.* [59] formalized the max-min fair objective under wireless scheduling constraints. They proposed a fair scheduling method which assigns dynamic weights to the flows such that the weights depend on the congestion in the neighborhood and schedules the flows which constitute a maximum weighted matching. Besides, many algorithms have been proposed to implement weighted max-min fairness with various degrees of accuracy, such as round-robin, proportional resource sharing [62] and weighted fair queuing [23]. These algorithms have been applied to allocating various types of resources. For example, Caprita *et al.* [16] and Stoica *et al.* [58] proposed proportional share resource allocation algorithms for CPU resources, Agrawala *et al.* [4] and Waldspurger *et al.* [62] studied fair queuing for memory resources, Bennett *et al.* [9] proposed an approximation algorithm based on the weighted fair queuing for allocating bandwidth resources.

2.1.3 Fairness for Multiple Types of Resources/Jobs

Ghodsi *et al.* [29] proposed a generalization of max-min fairness to multiple resource types known as Dominant Resource Fairness (DRF). DRF aims to maximize the minimum dominant share across all users, where the dominant share of a user is defined as the maximum share among the shares of all resource types that the user is allocated. Dolev *et al.* [24, 25] proposed "no justified complaints" as an alternative to DRF. Bonald *et al.* [13, 14] proposed Bottleneck Max Fairness to achieve a better efficiency-fairness tradeoff than DRF. Wang *et al.* [63, 64] extended DRF to deal with heterogeneous machines. All the above studies have focused on multi-resource allocation where jobs require multiple types of resources simultaneously for processing. Different from these studies, in our context of distributed job execution, jobs do not need to acquire resources from various machine clusters simultaneously. Different tasks of a job can be processed by different clusters in an asynchronous manner. Thus, the above multi-resource allocation policies are not directly applicable to distributed job execution.

Li *et al.* [44] proposed a greedy-based algorithm to show their approach to combining the efficiency and fairness. Jobs are classified into different types according to the completion time requirements and bandwidth demands, and are sent to different scheduler branches. Each scheduler branch processes a single type of jobs in a greedy manner. A Justice Evaluation Function is used to evaluate the fairness among different types of jobs [11]. In this thesis, we do not classify jobs and we evaluate the fairness of resource allocation by calculating the standard deviation among the resources allocated to different jobs.

In summary, although a large number of mechanisms have been proposed for fair job resource allocation, most existing mechanisms are primarily designed for a single machine or machine cluster only. To the best of our knowledge, the problem of fair resource allocation for distributed job execution across a set of machine clusters or datacenters remains largely open.

2.2 Efficiency

Earlier work on job scheduling typically studied a cluster of machines only. In cloud computing nowadays, jobs can be assigned to different datacenters for processing. In this section, we organize the literatures related to efficiency into three sections as follows.

2.2.1 Shortest Remaining Processing Time

SRPT (Shortest Remaining Processing Time), also named SRTF (Shortest Remaining Time First) is a classic job scheduling method where the system always executes the jobs with the smallest amount of processing time first until all the jobs are finished [2, 27, 57]. Donald [57] proved that for a work-conserving queuing system, SRPT minimizes the number of jobs in the queue and hence the average job response time. Banasal *et al.* [8] analyzed the unfairness of SRPT by comparing SRPT with Processor-Sharing scheduling. They concluded that the degree of unfairness under SRPT is small. SRPT has been applied to many applications, such as traffic control [5, 31, 46]. In these works, SRPT queues are used to help reduce traffic congestion and optimize traffic flow. In addition, SRPT queues have also been considered to study the fluid limits, where a fluid limit is a predicted approximate limiting criteria of a stochastic model [7, 26, 32].

Based on SRPT, many studies proposed more efficient methods to adapt it to different scheduling problems. For example, Davis *et al.* [22] presented feasibility tests for SRPT that enable this scheduling approach to be used for real-time systems. Harchol *et al.* [36] proposed a method based on SRPT for improving the performance of web servers processing static HTTP requests. Mohanty *et al.* [50] proposed an algorithm called Shortest Remaining Burst Round Robin that schedules processes with shortest remaining burst in a round robin manner using the dynamic time quantum. The algorithm was shown to perform better than round robin in terms of reducing the number of context switches, average waiting time and average turnaround time.

Some recent work studied distributed job execution among multiple geo-distributed datacenters. Hung *et al.* [40] developed scheduling algorithms to optimize the average job response time. Similar to SRPT, the main idea of their algorithms is to always devote the resources in all datacenters to the smallest job and complete it as quickly as possible. Thus, there is not really any fairness in the instantaneous resource allocation among the jobs. In addition, it was assumed that each task of a job can be executed at only a single site. When the data to be processed by the tasks of a job is possibly available at multiple sites rather than just a single site, job scheduling will be more complicated. Before using SRPT or its variants to schedule jobs, we first need to choose a site for each task to execute.

2.2.2 Rigid Job Scheduling for Multiple Sites

In a distributed cloud system, applications might span across multiple datacenters [33]. A user-requested job may constitute a series of components. The completion of the job means all the components of the job are finished. Some works assumed that each component of a job can only be processed by a particular datacenter. For example, Benoit *et al.* [10] studied a conventional problem of scheduling multiple applications which are made of independent tasks, and proposed several heuristics for the offline and online versions of the problem. Li [45] and Varalakshmi *et al.* [60] proposed several job scheduling algorithms which could guarantee the QoS requirements of the jobs. Ghanbari *et al.* [28] proposed a priority based job scheduling algorithm.

Im *et al.* [41] 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 [41], we focus on scheduling 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.

Monaldo *et al.* [49] studied minimizing the sum of weighted completion times in a concurrent open shop environment. In this model, each machine has the capability of processing one type of components. Each job has a weight and requires different types of processing components. A greedy program was constructed to solve this problem. Different from [49], in our model, each task of a job can be assigned to one of multiple sites for execution.

2.2.3 Flexible Job Scheduling for Multiple Sites

Some works assumed that each component of a job may choose one of several datacenters for processing through transfer. Hajjat *et al.* [35] proposed a system called *Dealer* to help optimize the job response times by picking better combinations of component replicas which could be located across multiple datacenters and could be put together to construct a full job. To complete the job, these components need to be transferred across datacenters. Since all the components could be transferred across datacenters, the *Dealer* estimates the transfer time for each combination of component replicas and finds a best combination to complete the job. Besides, such information is updated dynamically. Chang *et al.* [17] and Zheng *et al.* [38] proposed different heuristics to minimize the job completion times in a distributed system, assuming jobs could be transmitted across clusters. However, transferring a large amount of data across different datacenters can lead to enormous bandwidth costs and network congestion in distributed systems. Thus, it is important to run computing tasks close to their input data so as to reduce data

movement. Different from the above works, we do not consider the data transfer across datacenters. In our model, a job is composed of a set of tasks, but each task could be processed only in one or several datacenters without transferring the input data. Thus, to complete a job, we need to assign all the tasks of the job to different datacenters.

Khaled *et al.* [66] considered a conflict between fairness in scheduling and data locality. When a job is released in a cluster, the job will wait for its tasks to be assigned to the nodes in the cluster. An algorithm named delay scheduling was proposed that can temporarily relax fairness to improve locality by requiring jobs to wait for a scheduling chance on a node with local data. Different from [66], we regard data locality as a constraint of job execution. In our model, the tasks of each job can be assigned to one or more clusters with data locality. We study distributed job execution under data locality constraints.

Isard *et al.* [42] presented a graph-based framework for cluster scheduling with data locality constraints. They built a resource-sharing model where users could share the cluster and jobs could run concurrently. They also considered the fairness and efficiency of executing jobs. Different from our work, Isard *et al.* [42] addressed the problem with a graph data structure and developed a framework according to the graph. They focused on jobs sharing resources in one cluster, while we consider a model in which jobs compete for resources among multiple clusters.

Chapter 3

Task Assignment and Scheduling for Improving Efficiency of Distributed Job Execution

In this chapter, we study the task assignment and scheduling for improving efficiency of distributed job execution in which data to be processed by each job is possibly available at multiple locations.[1] 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 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.

3.1 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 any one of the sites where the data input is available. These sites are referred to as the *available sites* of the

[1]The work in this chapter has been published in Proceedings of the 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 726-735, 2022.

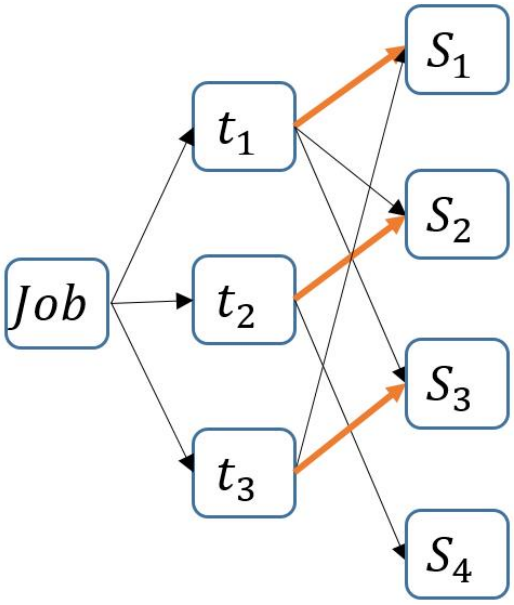


Figure 3.1: Example of available sites and processing sites

task. Figure 3.1 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.

A job is completed when all of its tasks are finished. The response time of a job refers to the duration from its release to its completion (i.e., the response time of a job is given by its completion time minus its release time). Given a sequence of jobs released over time, our objective is to schedule the job execution to minimize the average response time of all the jobs.

3.2 Job Scheduling 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 Sections 3.2.1 and 3.2.2, we propose two algorithms for

choosing the processing site of each task from its available sites. These two algorithms 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 3.2.3 and 3.2.4, we propose two holistic solutions that determine the processing sites and execution order of tasks together.

3.2.1 Balanced Task Allocation within 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. Our first algorithm, called Balanced Task Allocation within Jobs (BTAwJ), allocates tasks to sites for each job independently.

Given the available sites of the tasks in a 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 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 3.2) 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. The capacity of the edge from node S_j to the sink node is set to $u_j \cdot C$, 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

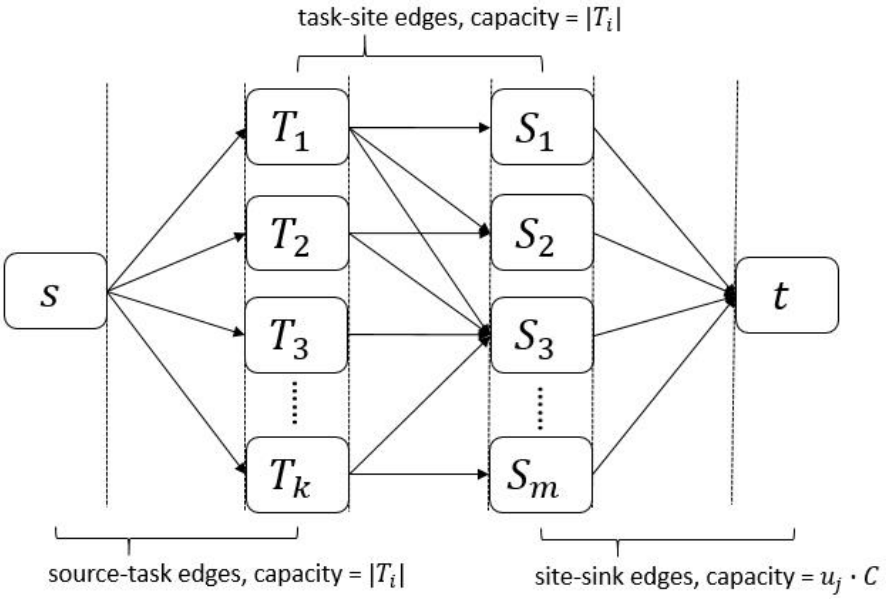


Figure 3.2: Flow network

C tasks per computing slot. In fact, given an integral flow f in the network, we can induce a task allocation as follows. For each task group T_i and each of its available sites S_i, 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 number of tasks assigned to each site must be capped at $u_j \cdot C$ since the edges connecting each node S_j to the sink node have the capacity $u_j \cdot C$. Vice versa, given a task allocation in which no site is allocated more than C tasks per computing slot, we can induce a flow in the flow network as follows. For each task group T_i and each of its available sites S_i, 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_i. 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 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. If the

Algorithm 1: Balanced Task Allocation within Jobs

Require: number of tasks in a job: n;
the available site set of each task in the job;
number of sites: m;
Ensure: allocation of the job's tasks to sites;
construct the flow network for the job;
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 $C_{upper} = \max_{1 \le k \le m} \lceil \frac{n}{u_j} \rceil$;
while $C_{lower} < C_{upper}$ **do**
  $C = \lfloor (C_{lower} + C_{upper})/2 \rfloor$;
  set the capacity of each edge (S_j, t) to $u_j \cdot C$;
  compute the maximum flow f of the network;
  if $|f| = n$ **then**
    $C_{upper} \leftarrow C$
  else
    $C_{lower} \leftarrow C + 1$;
  end if
end while
$C = C_{lower}$;
set the capacity of each edge (S_j, t) to $u_j \cdot C$;
compute the maximum flow of the network;
derive the task allocation of the job from the maximum flow;

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 there are n tasks in total and 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 tasks, each site S_j needs to run at most $\lceil \frac{n}{u_j} \rceil$ tasks per computing slot. Therefore, we can perform a binary search in the range $[\lceil \frac{n}{\sum_{j=1}^{m} u_j} \rceil, \max_{1 \le j \le m} \lceil \frac{n}{u_j} \rceil]$ 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. Algorithm 1 shows the details of the Balanced Task Allocation within Jobs (BTAwJ) algorithm.

We shall use a simple example of 3 jobs to illustrate our proposed scheduling solutions. Table 3.1 shows the arrival time and task number of each job. For simplicity, we assume

Table 3.1: 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	$\{S_2, S_3\}$

that all the tasks of a job have the same set of available sites. We also assume that each site has only one computing slot so that tasks are executed one at a time. The duration of each task is 1 second.

Figures 3.3, 3.4 and 3.5 show the task allocations for jobs J_1, J_2 and J_3 respectively by the BTAwJ algorithm. For J_1, since all of its tasks have the same available sites S_1 and S_2, BATwJ uniformly distributes the tasks between these two sites (see Figure 3.3). 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 3.4). 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 3.5). 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.

3.2.2 Balanced Task Allocation across Jobs

BTAwJ 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, by using BTAwJ, task allocation accumulated over multiple jobs may be quite unbalanced, which can adversely affect the job completion times. In the example of Figure 3.4, 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 the second algorithm, called Balanced

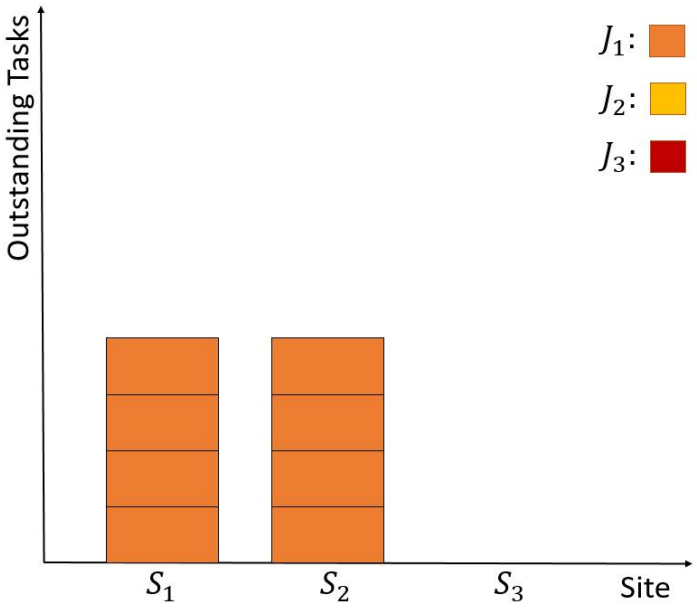


Figure 3.3: Outstanding tasks to execute at time $t = 0$ (s) by BTAwJ

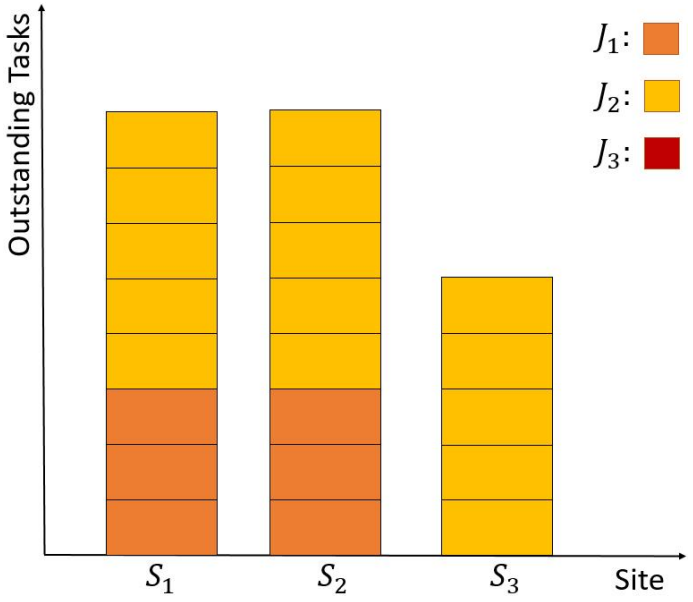


Figure 3.4: Outstanding tasks to execute at time $t = 1$ (s) by BTAwJ

Task Allocation across Jobs (BTAaJ), 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.

To do so, we can amend the flow network by adjusting the edge capacity between each node of a site and the sink node according to the remaining number of tasks to execute at the site. Specifically, let r_j denote the number of tasks already allocated to but not yet started at site S_j when a 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. In this way, we can establish a one-to-one correspondence between the integral flows in the network and the task allocation of the new job satisfying that the overall task number allocated to each site is at most C tasks per computing slot. Therefore, we can conduct a binary search to find the lowest C value to allocate all the tasks of the new job. Since there are a total number of $n+\sum_{j=1}^{m} r_j$ tasks for all the sites, a lower bound on the possible C values is $\lceil \frac{n}{\sum_{j=1}^{m} u_j} \rceil$. On the other hand, an upper bound on the possible C values is $\max_{1 \le j \le m} \lceil \frac{n+r_j}{u_j} \rceil$ since each site S_j needs to run at most $n+r_j$ tasks. Algorithm 2 shows the details of the Balanced Task Allocation across Jobs (BTAaJ) algorithm.

We use the same example of Table 3.1 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 3.3). 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 3.6). 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 3.7). 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 BTAwJ and BTAaJ algorithms, any scheduling algorithm can be used to determine the execution order of the tasks at each site.

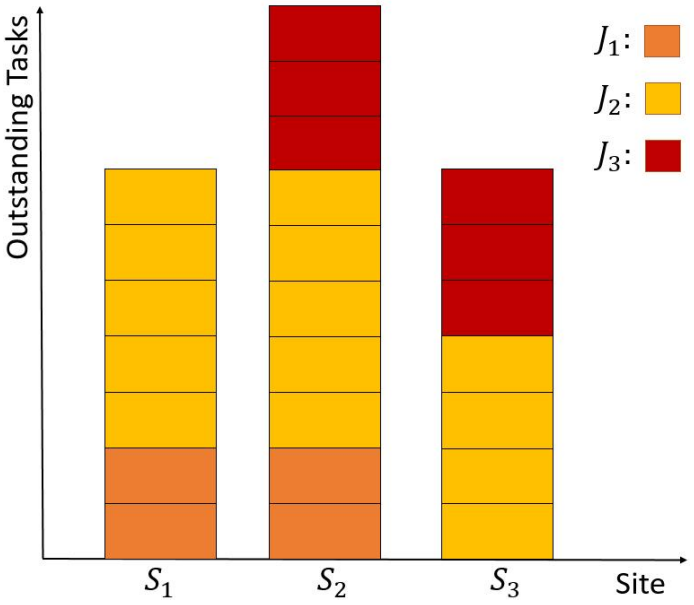


Figure 3.5: Outstanding tasks to execute at time $t = 2$ (s) by BTAwJ

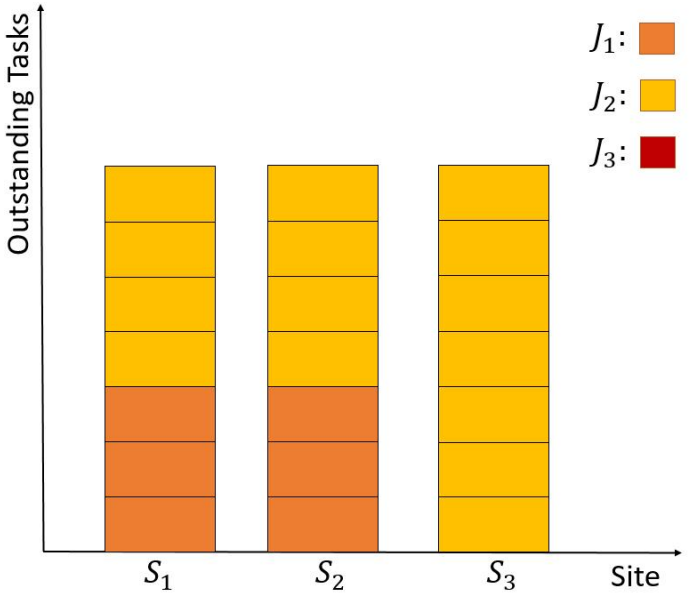


Figure 3.6: Outstanding tasks to execute at time $t = 1$ (s) by BTAaJ

Algorithm 2: Balanced Task Allocation across Jobs

Require: 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: $\{r_1, r_2, ..., r_m\}$;
Ensure: allocation of the new job's tasks to sites;
1: construct the flow network for the new job;
2: initialize the lower bound of C as $C_{lower} = \lceil \frac{n}{\sum_{j=1}^{m} u_j} \rceil$;
3: initialize the upper bound of C as $C_{upper} = \max_{1 \le j \le m} \lceil \frac{n+r_j}{u_j} \rceil$;
4: **while** $C_{lower} < C_{upper}$ **do**
5: $C = \lfloor (C_{lower} + C_{upper})/2 \rfloor$,
6: set the capacity of each edge (S_j, t) to $\max\{u_j \cdot C - r_j, 0\}$;
7: compute the maximum flow f of the network;
8: **if** $|f| = n$ **then**
9: $C_{upper} = C$;
10: **else**
11: $C_{lower} = C + 1$;
12: **end if**
13: **end while**
14: $C = C_{lower}$;
15: set the capacity of each edge (S_j, t) to $\max\{u_j \cdot C - r_j, 0\}$;
16: compute the maximum flow of the network;
17: derive the task allocation of the new job from the maximum flow;

3.2.3 Schedule Conscious Task Allocation

The above two task allocation algorithms are 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 execution. Hung *et al.* [40] 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. To the best of our knowledge, SWAG

is the state-of-the-art algorithm for the setting where each task of a job can be executed at only a single site. 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 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 $t_c + \max_{1 \leq j \leq m} \frac{\sum_{i=1}^{h} q_{i,j} + x_j}{u_j}$ (where t_c is the current time and u_j is the processing capacity of site S_j[2]) 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 $t_c + \max_{1 \leq j \leq m} \frac{\sum_{i=1}^{h} q_{i,j} + x_j}{u_j}$. This problem can again be modeled by the flow network constructed in Section 3.2.1. 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 $\max_{1 \leq j \leq m} \lceil \frac{n + \sum_{i=1}^{h} q_{i,j}}{u_j} \rceil$ respectively. Then, the lowest C value would be used as the estimated completion time

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

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 3 shows the details of the SCTA algorithm.

Again, we illustrate SCTA with the example of Table 3.1. 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 3.3). 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 3.6). 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 3.8). 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.

3.2.4 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 3.8). 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

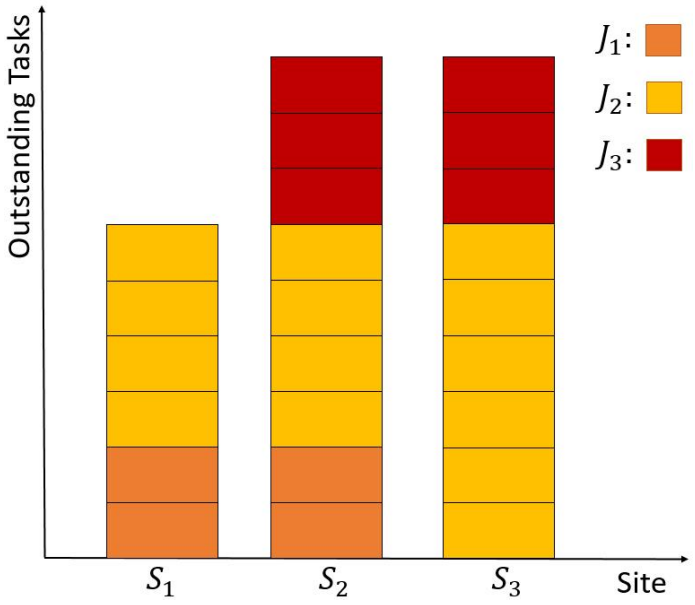


Figure 3.7: Outstanding tasks to execute at time $t = 2$ (s) by BTAaJ

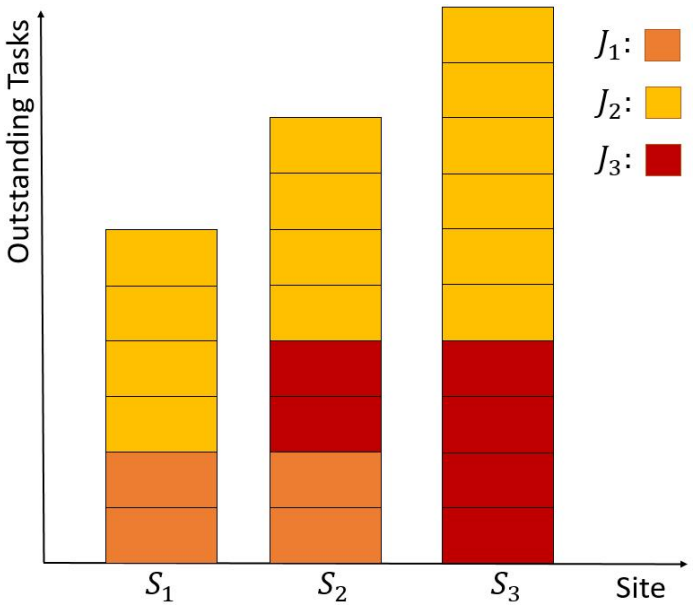


Figure 3.8: Outstanding tasks to execute at time $t = 2$ (s) by SCTA

Algorithm 3: Schedule Conscious Task Allocation

Require: 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};

Ensure: 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

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 3.8, 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 3.9). 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 3.2.3. Algorithm 4 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 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 3.10). 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 a decreasing 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

Algorithm 4: Adaptive Task Allocation

Require: 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};

Ensure: 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

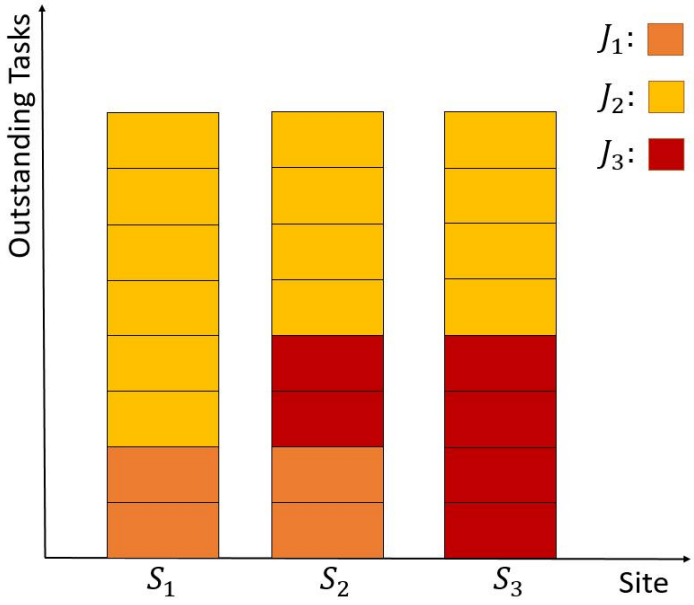


Figure 3.9: Outstanding tasks to execute at time $t = 2$ (s) by ATA

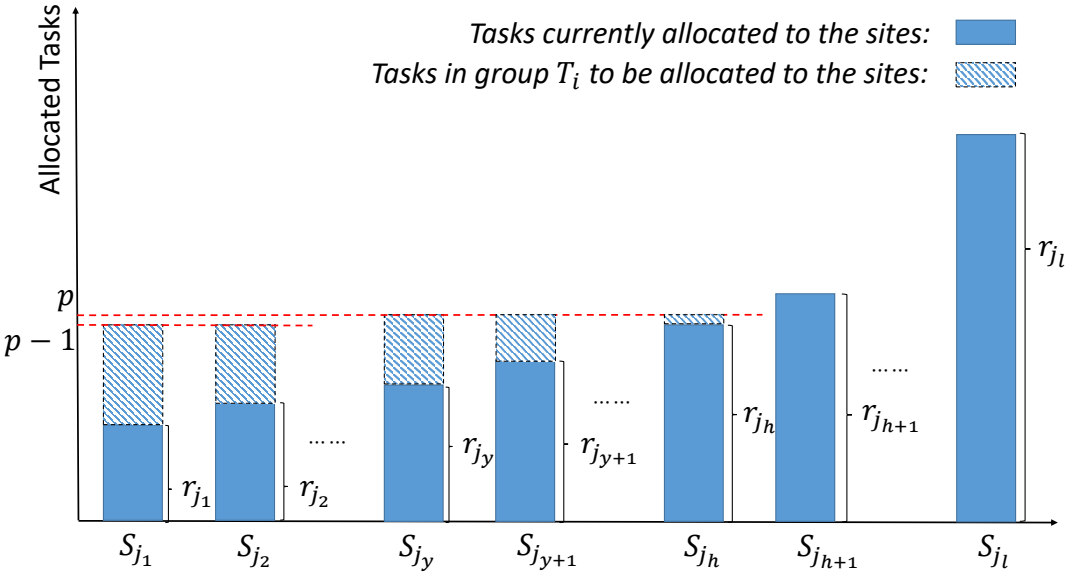


Figure 3.10: Task Assignment for ATA-Greedy

$r_{j_1}, r_{j_2}, ..., r_{j_l}$ denote the numbers of tasks currently allocated to these sites. Without loss of generality, assume that the available sites are sorted in an increasing order of their current workloads, *i.e.* $r_{j_1} \leq r_{j_2} \leq ... \leq r_{j_l}$. Define $r_{j_{l+1}} = \infty$. Let $|T_i|$ denote the number of tasks 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 $\sum_{x=1}^{h-1}(r_{j_h} - r_{j_x}) < |T_i| \leq \sum_{x=1}^{h}(r_{j_{h+1}} - r_{j_x})$. Then, the tasks in group T_i would be allocated to sites $S_{j_1}, S_{j_2}, ..., S_{j_h}$. Let p (where $r_{j_h} < p \leq r_{j_{h+1}}$) 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 $1 \leq x \leq y$, and allocates $p - r_{j_x}$ tasks to each site S_{j_x} where $x > y$. 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.

3.2.5 Discussion

In our proposed solutions, we simply use the number of remaining tasks to execute as an estimate of the job completion time. We do not assume or 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 [6]. 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.

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 [40]. Thus, in this case, our solutions can primarily be used to assign and schedule tasks in the first stage.

Table 3.2 summarizes the five algorithms proposed in this section and their relations. We can see that from BTAaJ to ATA, the algorithms gradually take more factors into consideration in the task allocation to improve scheduling performance. Meanwhile, the computational complexity increases as more factors are considered. Thus, ATA-Greedy

is proposed for reducing the computational complexity of ATA while maintaining its characteristics.

Table 3.2: Summary of Algorithms

Algorithms	**Characteristics**
BTAaJ	Task allocation of a new job is determined within the job.
BTAwJ	Task allocation of a new job is determined considering all the outstanding jobs in the system.
SCTA	Task allocation of a new job is determined considering all the outstanding jobs in the system and exploring the prioritization of jobs.
ATA	Task allocation of a new job and existing jobs are determined considering all the outstanding jobs in the system and exploring the prioritization of jobs.
ATA-Greedy	A heuristic of ATA which reduces the computational complexity.

Our proposed algorithms are heuristics. They use the number of remaining tasks to execute as an estimate of the job completion time. Since they do not make use of any knowledge on task durations, they are unlikely to have approximation bounds in terms of the average job response time. Nevertheless, it is possible to derive some rough approximation bounds under restricted assumptions. Suppose that the tasks of all jobs have identical durations. Then, the remaining time to job completion is proportional to the number of remaining tasks. In this special case, at each new job arrival, the task allocations and schedules computed by the SCTA and ATA algorithms (Algorithms 3 and 4) have an approximation ratio of m in terms of the average job response time, where m is the number of sites and the response time of each job is measured by the job completion time minus the current time (*i.e.*, the time of the new job arrival). For simplicity, assume that the current time is 0. Our analysis below is inspired by the work of [43] on the order scheduling model.

Following the notations in Algorithms 3 and 4, suppose that at the arrival of a new job, there are a total of $g+1$ outstanding jobs (including the new job): $J_1, J_2, \ldots, J_{g+1}$. For each job J_i, let $(q_{i,1}, q_{i,2}, \ldots, q_{i,m})$ denote the task allocation of J_i among the m sites finally computed by the SCTA or ATA algorithm, where $q_{i,j}$ is the number of tasks

assigned to site S_j and $\sum_{j=1}^{m} q_{i,j}$ is the total number of remaining tasks for J_i. Let $Q_j = \max\{q_{i,1}, q_{i,2}, \ldots, q_{i,m}\}$ be the maximum number of tasks allocated to a site for job J_i among all sites. Without loss of generality, assume that the jobs are indexed in increasing order of Q_j, *i.e.*, $Q_1 \leq Q_2 \leq \cdots \leq Q_{g+1}$. Let $J_{p_1}, J_{p_2}, \ldots, J_{p_{g+1}}$ denote the order of job execution finally computed by the SCTA or ATA algorithm, where J_{p_1} is the first job to execute, $J_{p_{g+1}}$ is the last job to execute, and $p_1, p_2, \ldots, p_{g+1}$ is a permutation of $1, 2, \ldots, g+1$ (note that p_i and i are not necessarily the same number). Without loss of generality, assume that the duration of each task is one time unit. Let $C(J_{p_i})$ denote the completion time of job J_{p_i} by the SCTA or ATA algorithm.

Recall that following the SWAG algorithm, in each iteration of the SCTA and ATA algorithms, we estimate the completion times of all the jobs yet to be ordered and select the job with the earliest completion time to append to the job order. Thus, the first job appended to the job order must be J_1, *i.e.*, $p_1 = 1$. As a result, $C(J_{p_1}) = Q_1$, since a job is completed when all of its tasks are finished. Subsequently, at the beginning of each iteration $i \geq 2$, the job order contains $i-1$ jobs only, so at least one of the jobs $J_1, J_2, \ldots, J_i$ have not yet been added to the job order. Suppose that J_x (where $1 \leq x \leq i$) have not been added. If J_x is appended to the job order, its completion time is at most $C(J_{p_{i-1}}) + Q_x$. This is because in the SCTA algorithm, the task allocation of the job is computed at its arrival and it does not change afterwards. So, if J_x is an existing job, its allocation must be the same as its final allocation $(q_{x,1}, q_{x,2}, \ldots, q_{x,m})$ in each iteration. Since each site is assigned at most Q_x additional tasks, J_x's completion time is at most $C(J_{p_{i-1}})+Q_x$ if it is appended to the job order. If J_x is the incoming new job, its allocation computed in each iteration may not be the same as its final allocation $(q_{x,1}, q_{x,2}, \ldots, q_{x,m})$. Since the maximum flow computation finds the best task allocation given the jobs already in the job order, if the new job J_x is appended to the job order, its completion time is still bounded by that resulting from the final allocation $(q_{x,1}, q_{x,2}, \ldots, q_{x,m})$. Thus, J_x's completion time is bounded by $C(J_{p_{i-1}}) + Q_x$. On the other hand, in the ATA algorithm, the task allocations of all the existing and new jobs can be adjusted. For each job J_x, its allocation computed in each iteration may not be the same as its final allocation $(q_{x,1}, q_{x,2}, \ldots, q_{x,m})$. Similarly, since the maximum flow computation finds the best task allocation given the jobs already in the job order, if J_x is appended to the

job order, its completion time is still bounded by that resulting from the final allocation $(q_{x,1}, q_{x,2}, \ldots, q_{x,m})$. Hence, J_x's completion time is bounded by $C(J_{p_{i-1}}) + Q_x$. By the greedy strategy of the SCTA and ATA algorithms, the actual job J_{p_i} appended to the job order must lead to its completion time no later than $C(J_{p_{i-1}}) + Q_x \leq C(J_{p_{i-1}}) + Q_i$. Therefore, we have $C(J_{p_i}) - C(J_{p_{i-1}}) \leq Q_i$. So, the total job completion time by the SCTA or ATA algorithm satisfies

$$\sum_{i=1}^{g+1} C(J_{p_i}) = \sum_{i=1}^{g+1} \left(C(J_{p_1}) + \sum_{k=2}^{i} \left(C(J_{p_k}) - C(J_{p_{k-1}}) \right) \right) \leq \sum_{i=1}^{g+1} \sum_{k=1}^{i} Q_k. \tag{3.1}$$

Now consider the optimal task allocation and schedule that minimizes the average job response time. For each job J_i, let $(q^*_{i,1}, q^*_{i,2}, \ldots, q^*_{i,m})$ denote the task allocation of J_i among the m sites in the optimal solution, where $q^*_{i,j}$ is the number of tasks assigned to site S_j and $\sum_{j=1}^{m} q^*_{i,j}$ is the total number of remaining tasks for J_i. Since the total number of remaining tasks for each job is fixed, we have

$$\sum_{j=1}^{m} q^*_{i,j} = \sum_{j=1}^{m} q_{i,j}. \tag{3.2}$$

Let $J_{o_1}, J_{o_2}, \ldots, J_{o_{g+1}}$ denote the order of job execution in the optimal solution, where J_{o_1} is the first job to execute, $J_{o_{g+1}}$ is the last job to execute, and $o_1, o_2, \ldots, o_{g+1}$ is a permutation of $1, 2, \ldots, g+1$ (note that o_i and i are not necessarily the same number). Let $C^*(J_{o_i})$ denote the completion time of job J_{o_i} in the optimal solution. Then, it holds

that

$$\begin{aligned}
C^*(J_{o_i}) &= \max_{1\le j\le m}\{\sum_{k=1}^{i} q^*_{o_k,j}\} \\
&\ge \frac{\sum_{j=1}^{m}(\sum_{k=1}^{i} q^*_{o_k,j})}{m} \\
&= \frac{\sum_{k=1}^{i}(\sum_{j=1}^{m} q^*_{o_k,j})}{m} \\
&= \frac{\sum_{k=1}^{i}(\sum_{j=1}^{m} q_{o_k,j})}{m} \\
&\ge \frac{\sum_{k=1}^{i} \max_{1\le j\le m}\{q_{o_k,j}\}}{m} \\
&= \frac{\sum_{k=1}^{i} Q_{o_k}}{m} \\
&\ge \frac{\sum_{k=1}^{i} Q_k}{m},
\end{aligned}$$

where the third equality above is due to equation (3.2) and the last inequality is based on the assumption that $Q_1 \le Q_2 \le \cdots \le Q_{g+1}$. It follows that the total job completion time by the optimal solution satisfies

$$\sum_{i=1}^{g+1} C^*(J_{o_i}) \ge \sum_{i=1}^{g+1}((\sum_{k=1}^{i} Q_k)/m) = (\sum_{i=1}^{g+1}\sum_{k=1}^{i} Q_k)/m. \tag{3.3}$$

By equations (3.1) and (3.3), the total job completion time and hence the average job response time by the SCTA and ATA algorithms are at most m times of the optimal.

3.3 Experimental Setup

We conduct 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 [3, 20], 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 $\beta = 1.259$ [6] and a mean of 2 seconds. The Google trace was collected on a cluster of about 12000 machines for one month at Google [1,53]. 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. The job arrival times are available in the original traces. We increase or reduce the inter-arrival times of the jobs in the traces proportionally to simulate different levels of system utilization from 40% to 70%.

Site Capacity: The default number of sites is set at 10. The sites are denoted as S_1, S_2, ..., S_{10}. 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 set the number of available sites at 2.

Scheduling Methods: We implement all the scheduling solutions described in Section 3.2. For the BTAwJ and BTAaJ algorithms, 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, since to our knowledge, it is the state-of-the-art algorithm for optimizing the average job response time. 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.

3.4 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 3.11 and 3.12 show the results of Google and Facebook traces respectively with each task having 2 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.

As can be seen from Figures 3.11 and 3.12, the SCTA and ATA algorithms outperform the BTAwJ and BTAaJ algorithms. This indicates that the solutions combining task allocation and job scheduling strategies are more effective for optimizing job response times than the solutions 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 demonstrate that ATA is more capable in 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 3.13 and 3.14 show

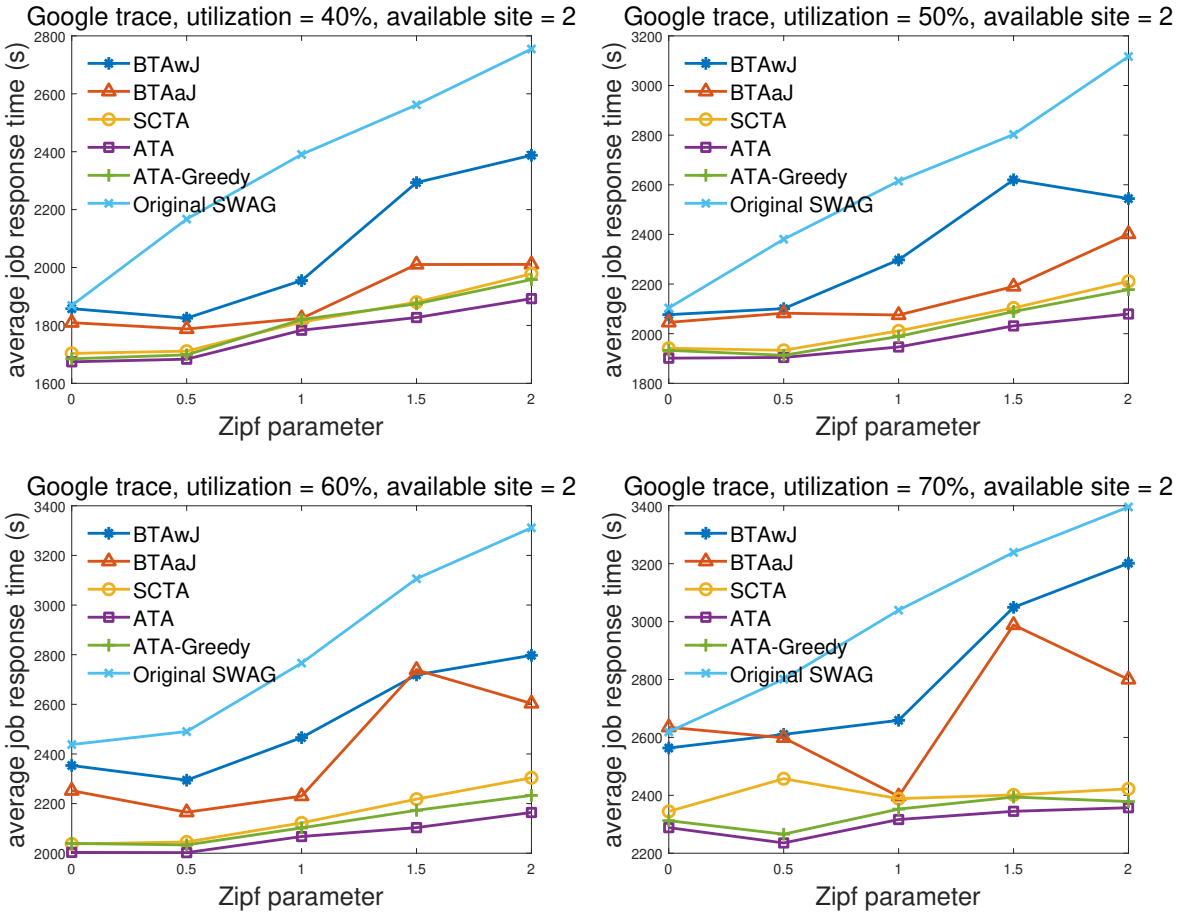


Figure 3.11: Average job response time for Google trace (2 available sites for each task)

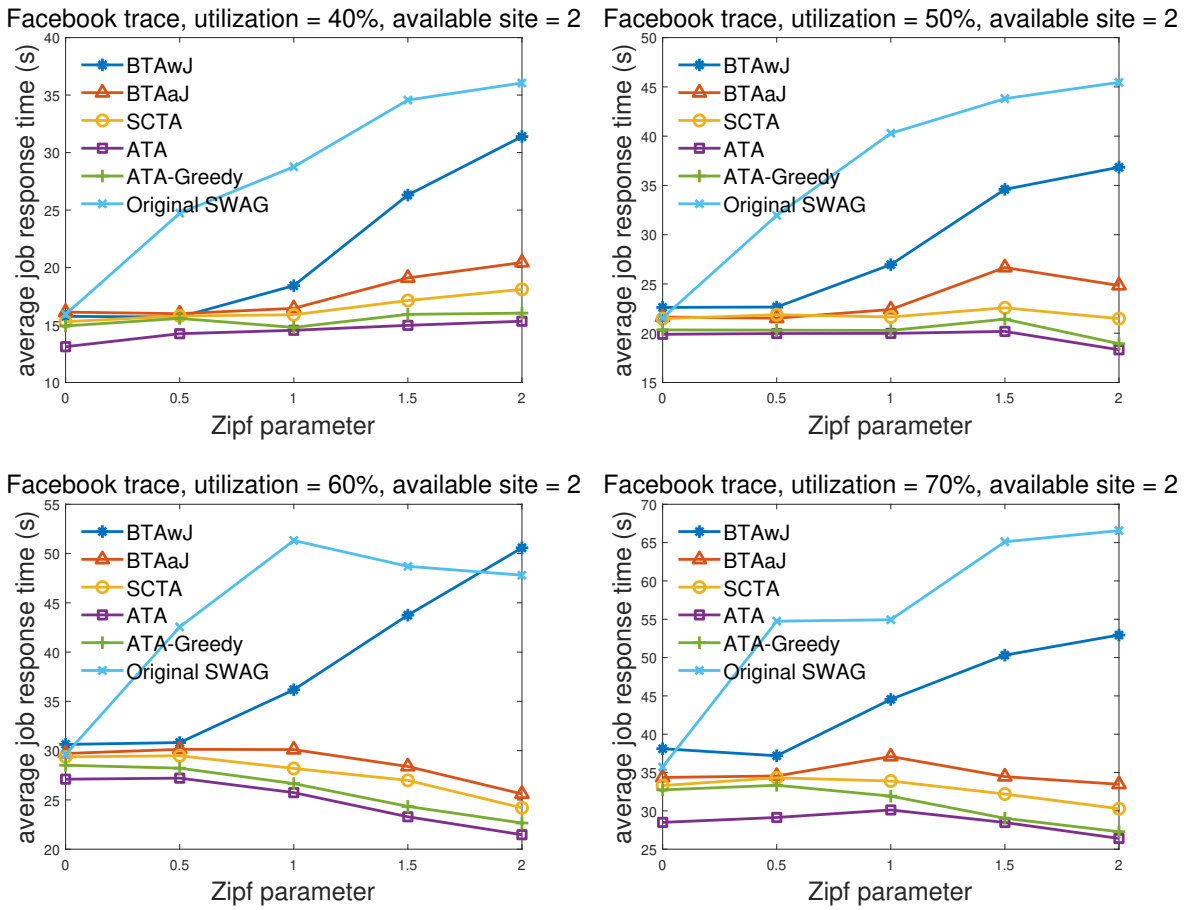


Figure 3.12: Average job response time for Facebook trace (2 available sites for each task)

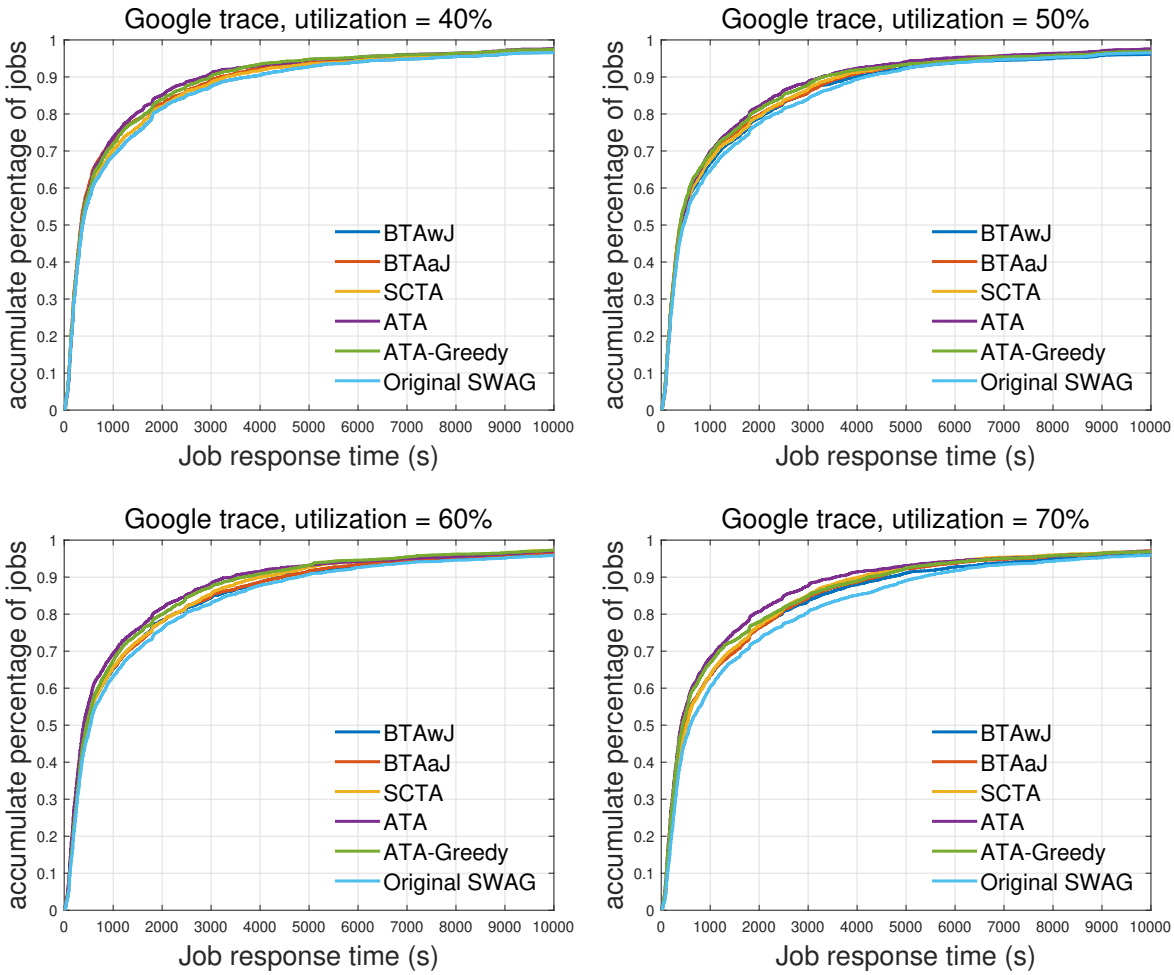


Figure 3.13: Cumulative distribution of job response time for Google trace (Zipf parameter = 1)

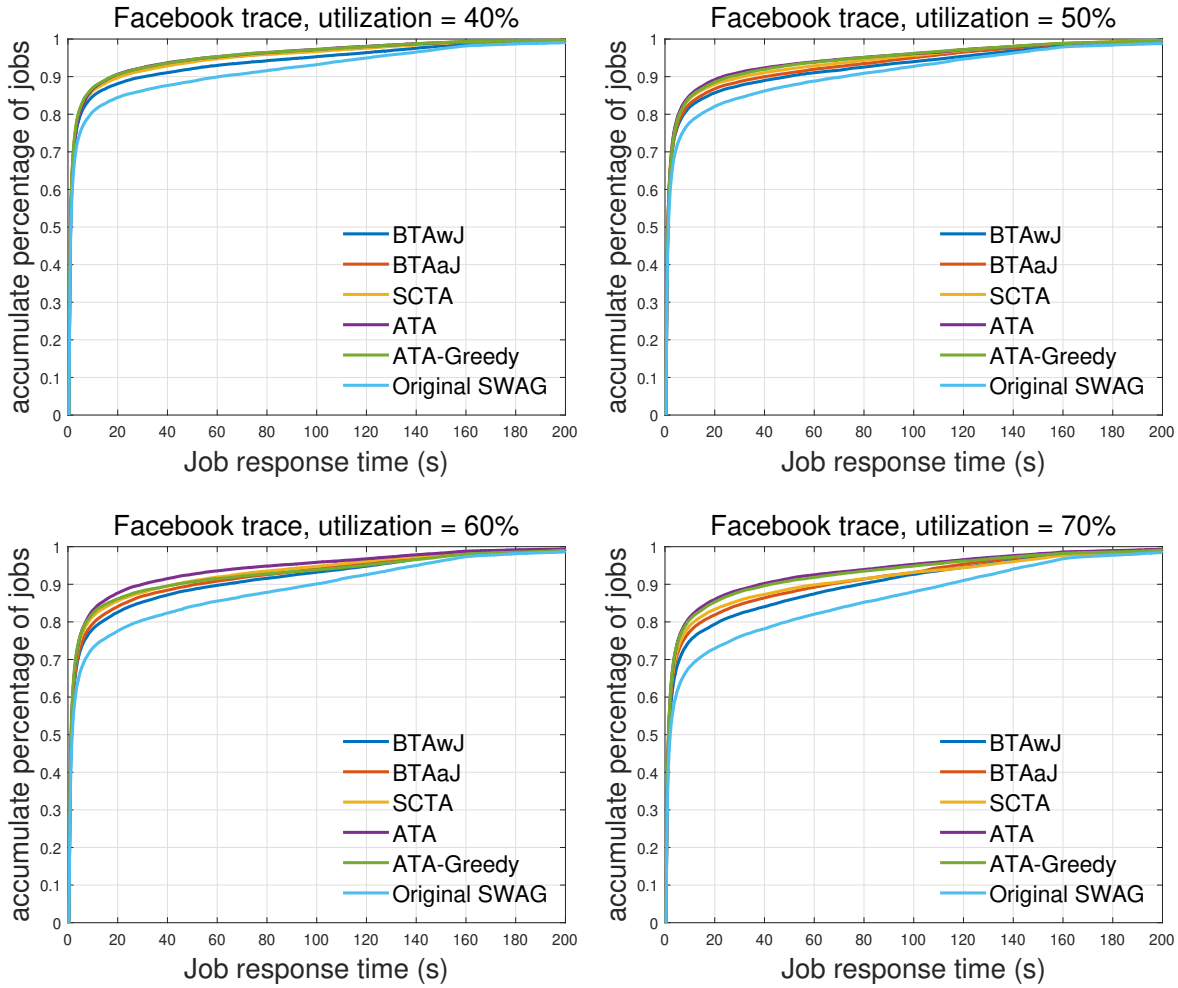


Figure 3.14: Cumulative distribution of job response time for Facebook trace (Zipf parameter = 1)

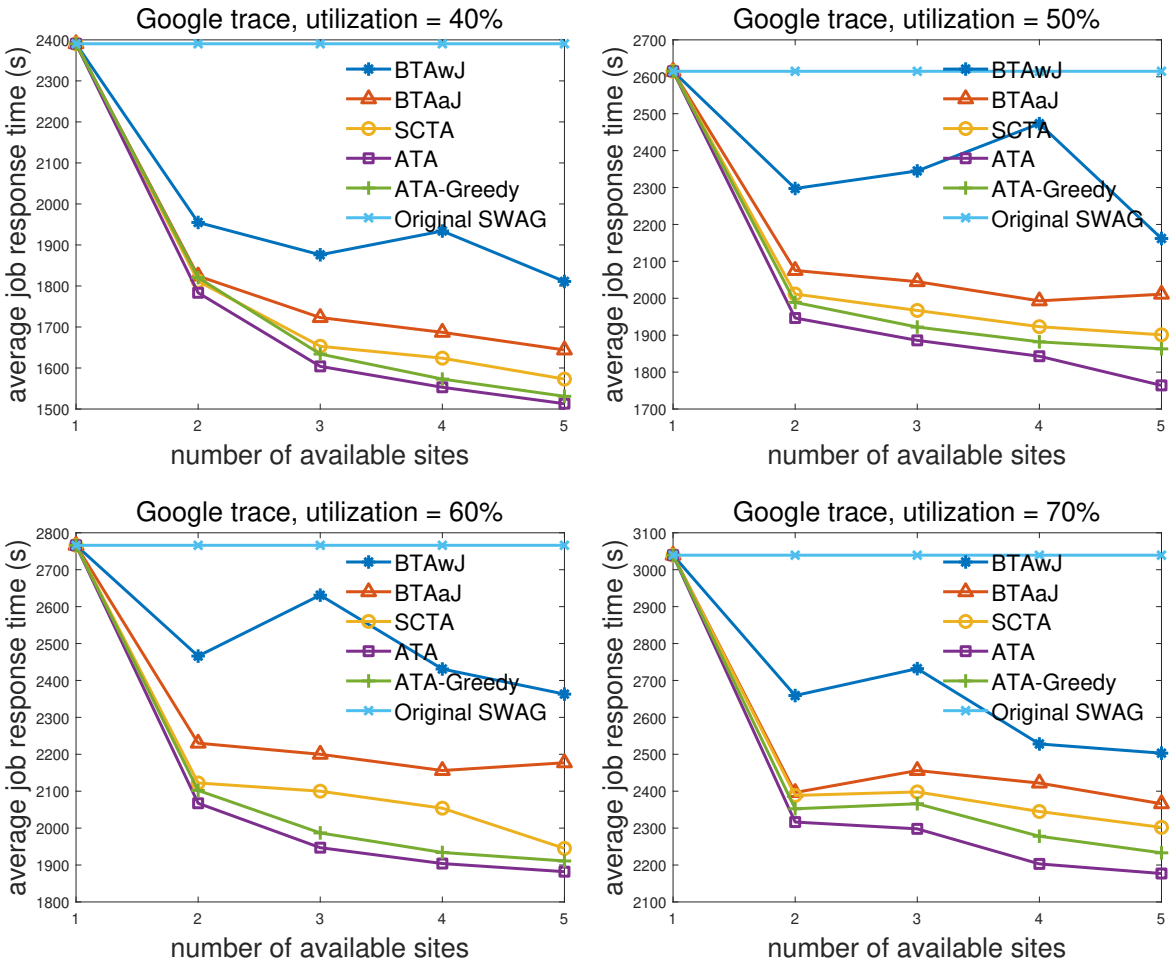


Figure 3.15: Average job response time for Google trace (Zipf parameter = 1)

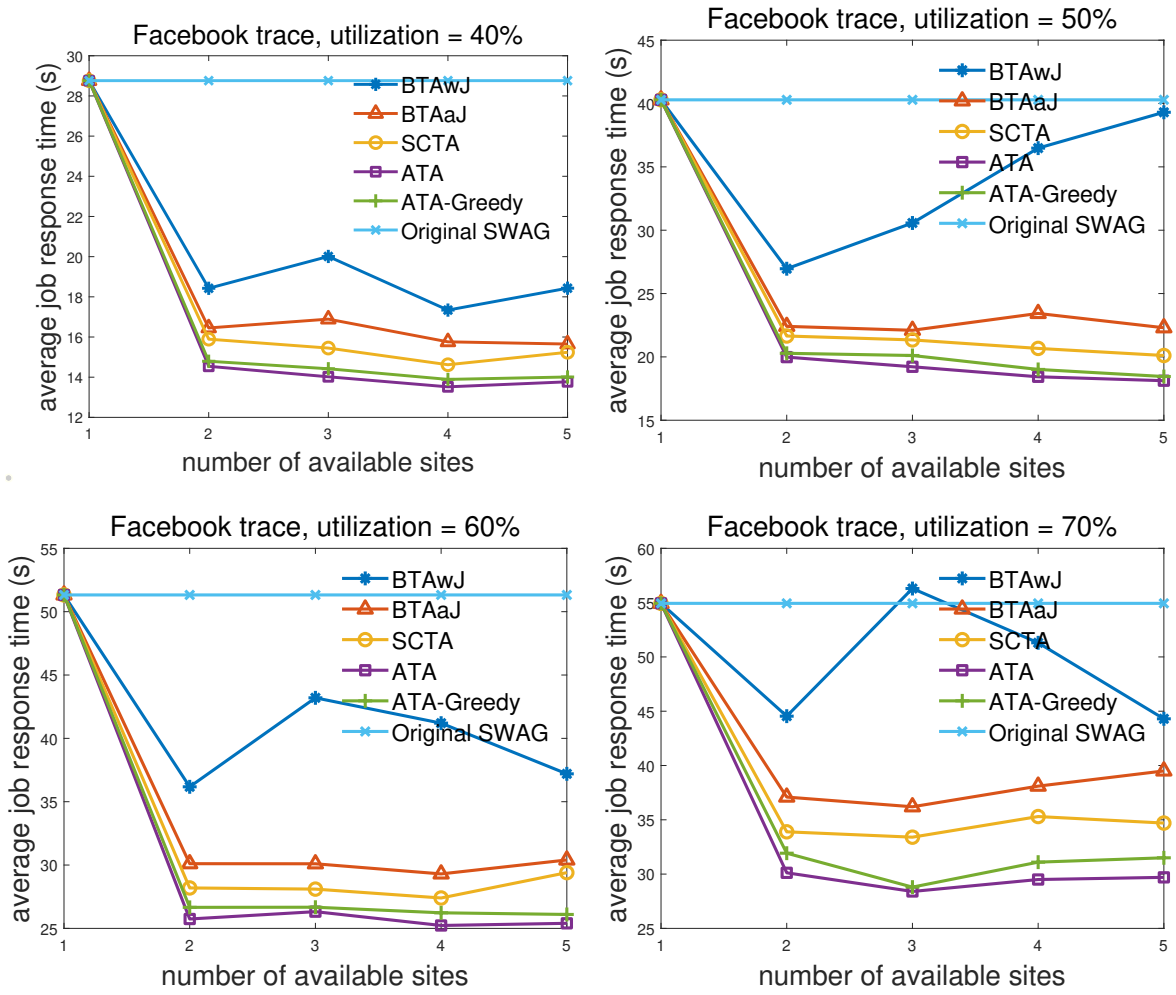


Figure 3.16: Average job response time for Facebook trace (Zipf parameter = 1)

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.

The original SWAG algorithm allocates each task to a fixed available site and does not make use of other available sites. Thus, it normally has the most skewed task allocation and results in much higher job response times than all our proposed algorithms except BTAwJ. The relative performance of BTAwJ to the original SWAG algorithm varies over different settings. As seen from Figures 3.11 and 3.12, BTAwJ performs even worse than SWAG in terms of average job response time for some settings. This is because BTAwJ conducts task assignment independently for each job. BTAwJ simply distributes the tasks evenly among the available sites for each job and does not consider the number of outstanding tasks (of other jobs) to execute at each site. As a result, the instantaneous task queue lengths at various sites can be quite different and even less balanced than those in the original SWAG algorithm, giving rise to possibly longer job response times.

Figures 3.15 and 3.16 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 response times. The relative performance of the algorithms remains largely unchanged for different numbers of available sites.

3.5 Summary

In this chapter, we have studied task assignment and scheduling for improving efficiency of distributed job execution in which each task of a job may be executed at a subset of all the sites. The assignment of each task to one of its available sites gives an additional

dimension of freedom in resource allocation, which makes the design of solutions more complicated than the setting where each task can be executed at only a single site. 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 as well as a baseline that allocates each task to a fixed available site. Among the five algorithms proposed in this chapter, the experimental results show that the ATA and ATA-Greedy algorithms have similar performance and they both outperform the other algorithms. Since ATA-Greedy has a lower computational complexity than ATA, ATA-Greedy is more suitable for practical use.

Chapter 4

Max-min Fair Resource Allocation for Distributed Job Execution

Data analytic jobs usually require large volumes of data inputs that are available at geographically distributed locations. Gathering all the data at a central location for processing would not only place heavy traffic burdens on the underlying networks but also slow down the job execution. To achieve better performance, it is often preferable to take advantage of data locality by distributing job execution across multiple sites located close to the data to be processed [51, 61]. When there are not enough resources to fully meet the demands of all the jobs at one or more sites, how to allocate resources fairly among the jobs is a critical problem. In the literature, a large number of mechanisms have been proposed for fair resource allocation as discussed in Chapter 2. However, most existing mechanisms are primarily designed for a single machine or machine cluster only. To the best of our knowledge, no work has considered fair resource allocation in the setting of distributed job execution. In this chapter, we study the problem of fair resource allocation for distributed job execution across a set of machine clusters or datacenters.[1] This is a challenging problem in that different jobs may require different combinations of sites for execution and may have different resource demands at each site.

[1]The work in this chapter has been published in Proceedings of the 48th International Conference on Parallel Processing. article no. 55, pp. 1-10, 2019.

4.1 Preliminary: Max-Min Fairness

Max-min fairness is a well-recognized approach to define fairness in resource allocation [12]. Generally speaking, max-min fairness aims to allocate resources as much as possible to jobs with low demands, and equally distribute resources among the remaining jobs whose demands cannot be completely satisfied. Bertsekas *et al.* [12] gave a formal and generic definition of max-min fairness:

Definition 4.1 ([12]). *Consider a set X of n-dimensional vectors. A vector $\vec{x} \in X$ is max-min fair on set X, if and only if, $\forall \vec{y} \in X$, if $y_s > x_s$ holds for an index $s \in \{1, 2, \ldots, n\}$, then there must exist another index $t \in \{1, 2, \ldots, n\}$ such that $y_t < x_t \leq x_s$.*

Definition 4.1 implies that in a max-min fair vector $\vec{x}$, increasing any component x_s must be done at the expense of decreasing another smaller or equal component x_t. Bertsekas *et al.* [12] proved that if a max-min fair vector exists on a set X, then it is unique. A follow-up work by Radunovic and Le Boudec [52] studied the achievability of max-min fairness. A set X is said max-min achievable if there exists a max-min fair vector on X. Radunovic and Le Boudec [52] identified and proved a sufficient condition for a set X to be max-min achievable:

Theorem 4.1 ([52]). *If a set X is compact and convex, then it is max-min achievable.*

By viewing resource allocation among jobs as a vector, it is straightforward to instantiate Definition 4.1 to define the max-min fairness of resource allocation among multiple jobs running on a single machine. To illustrate, consider four jobs running on a machine whose resource capacity is 20. Suppose the resource demands of the four jobs are described by a vector $\langle 2, 4, 10, 40 \rangle$. Clearly, the machine cannot fully meet the demands of all jobs. An allocation vector $\langle 2, 3, 7, 8 \rangle$ does not satisfy max-min fairness, because it is possible to increase job 2's allocation of 3 by decreasing job 3's allocation of 7 which is larger than 3. The allocation vector satisfying max-min fairness is $\langle 2, 4, 7, 7 \rangle$. With this allocation, no component can be increased without decreasing an equal or even smaller component.

Max-min fairness is always achievable for resource allocation on a single machine. It also has some elegant properties: Pareto efficiency, envy-freeness, strategy-proofness and sharing incentive, which are widely accepted notions of fairness [29].

Pareto efficiency: It is not possible to increase the allocation of a job without decreasing the allocation of another job. This property implies that resource utilization is maximized. In particular, all the resources must be allocated unless the total resource demand is below the resource availability.

Envy-freeness: A job would not prefer the allocation of any other job. This is a common requirement for fairness.

Strategy-proofness: A job cannot increase its allocation by lying about its demands. This property provides incentive compatibility.

Sharing incentive: Each job should be better off sharing the resources, compared to statically partitioning the resources and exclusively using its own partition of the resources.

4.2 System Model

We consider a distributed system consisting of a set of separate sites: $\mathcal{M} = \{S_1, S_2, ..., S_m\}$. Each site can refer to a cluster of machines or a datacenter. Each site S_j has a resource capacity u_j which can be understood as the number of computing slots available at the site.

Suppose there is a set of n jobs $\mathcal{J} = \{J_1, J_2, ..., J_n\}$ running in the system. Each job J_i is composed of tasks that process independent data partitions and can run in parallel. We assume that each task is to be executed at a designated site in the set $\mathcal{M}$ due to reasons such as data locality (each task is assigned to the site that holds the input data) and unacceptable overheads of data migration between sites. A job J_i's resource demand at a site is given by the total demand of its tasks to be executed at the site. For example, if each task occupies one computing slot during execution, then at any time, the remaining number of job J_i's tasks to be executed at a site can be understood as J_i's resource demand at the site. We model the resource demands of all jobs as a $n \times m$ matrix $\mathbf{D}_{n\times m}$, in which each entry d_{ij} represents job J_i's resource demand at site S_j. If

J_i has no task to run at a site S_j, we define $d_{ij} = 0$. A job is completed only when all of its tasks are finished.

Resource allocation among the jobs at different sites can be represented by a $n \times m$ matrix $\mathbf{A}_{n\times m}$, in which each entry a_{ij} indicates the amount of resources allocated to job J_i by site S_j. We refer to $\mathbf{A}_{n\times m}$ as an allocation matrix or an allocation for short. Naturally, any *feasible* allocation must satisfy the site capacity constraints:

$$\forall S_j \in \mathcal{M}, \quad \sum_{i=1}^{n} a_{ij} \leq u_j, \tag{4.1}$$

which means that the total amount of resources allocated by each site S_j cannot exceed its capacity. On the other hand, it does not make sense to allocate more resources than what a job needs at any site. Thus, any *rational* allocation should satisfy the job demand constraints:

$$\forall J_i \in \mathcal{J}, S_j \in \mathcal{M}, \quad 0 \leq a_{ij} \leq d_{ij}. \tag{4.2}$$

We consider feasible and rational allocations only, and focus on the fairness of resource allocation among the jobs.

4.3 Max-min Fairness Across Multiple Sites

A naive extension of max-min fairness to distributed job execution is to consider each site independently and require the resource allocation at each site to be max-min fair. We refer to this policy as Independent Max-min Fairness (*IMF*). By the results of max-min fairness on a single machine, it is straightforward that *IMF* is achievable and satisfies all the four properties presented in Section 4.1.

By definition, for any site, if every job has sufficiently large resource demand at the site, *IMF* assigns each job an equal share of the resources at the site. However, in our context of distributed job execution, a job does not necessarily have tasks to run at every site since the task assignment is determined by data availability. Thus, the aggregate resources allocated to different jobs by *IMF* can be far from equal or fair. In this section, we propose a new resource allocation policy called Aggregate Max-min Fairness for distributed job execution.

4.3.1 Aggregate Max-min Fairness

Aggregate Max-min Fairness (*AMF*) considers all the sites as a united resource pool and defines fairness according to the total amount of resources acquired by each job from all the sites. The rationale is that in distributed job execution, the system-wide processing rate of each job is decided by its aggregate resource allocation. Given an allocation matrix $\mathbf{A}_{n\times m}$, we define a job-wise allocation vector $\vec{A}$ to describe the aggregate amount of resources received by each job from all the sites:

$$\vec{A} = \langle \sum_{j=1}^{m} a_{1j}, \sum_{j=1}^{m} a_{2j}, ..., \sum_{j=1}^{m} a_{nj} \rangle.$$

AMF requires that the vector $\vec{A}$ is max-min fair. Formally, let $\mathcal{X}$ denote the set of all the feasible and rational allocation matrices. Let X be the set including all the job-wise allocation vectors of the allocation matrices in $\mathcal{X}$.

Definition 4.2. *An allocation $\mathbf{A}_{n\times m}$ satisfies Aggregated Max-min Fairness, if and only if, its job-wise allocation vector $\vec{A}$ is the max-min fair vector over the set X.*

According to Theorem 4.1, if the set X is compact and convex, then *AMF* is achievable.

Lemma 4.1. *The set X is compact.*

Proof. Consider any job-wise allocation vector

$$\vec{A} = \langle \sum_{j=1}^{m} a_{1j}, \sum_{j=1}^{m} a_{2j}, ..., \sum_{j=1}^{m} a_{nj} \rangle$$

produced by a feasible and rational allocation matrix $\mathbf{A}_{n\times m}$. From the demand and capacity constraints (*i.e.*, (4.1) and (4.2)), for each job J_i, $0 \leq \sum_{j=1}^{m} a_{ij} \leq \sum_{j=1}^{m} \min\{u_j, d_{ij}\}$. Thus, the set X is closed and bounded. Hence, it is compact. □

Lemma 4.2. *The set X is convex.*

Proof. We prove that for any two vectors $\vec{A}, \vec{B} \in X$ and any $0 \leq \lambda \leq 1$, it holds that $\lambda\vec{A} + (1-\lambda)\vec{B} \in X$.

Suppose that $\mathbf{A}_{n\times m}$ and $\mathbf{B}_{n\times m}$ are two allocations producing $\vec{A}$ and $\vec{B}$ respectively. Consider the allocation $\mathbf{C}_{n\times m} = \lambda\mathbf{A}_{n\times m} + (1-\lambda)\mathbf{B}_{n\times m}$. It is obvious that the job-wise allocation vector produced by $\mathbf{C}_{n\times m}$ is $\lambda\vec{A} + (1-\lambda)\vec{B}$.

Since $\mathbf{A}$ and $\mathbf{B}$ are rational, by the demand constraints (4.2), we have $0 \leq \lambda a_{ij} \leq \lambda d_{ij}$ and $0 \leq (1-\lambda)b_{ij} \leq (1-\lambda)d_{ij}$. Thus, $0 \leq \lambda a_{ij} + (1-\lambda)b_{ij} = c_{ij} \leq d_{ij}$. Similarly, since $\mathbf{A}$ and $\mathbf{B}$ are feasible, by the capacity constraints (4.1), we have $0 \leq \lambda \sum_{i=1}^{n} a_{ij} \leq \lambda u_j$ and $0 \leq (1-\lambda)\sum_{i=1}^{n} b_{ij} \leq (1-\lambda)u_j$. It follows that $0 \leq \sum_{i=1}^{n} \lambda a_{ij} + \sum_{i=1}^{n}(1-\lambda)b_{ij} = \sum_{i=1}^{n} c_{ij} \leq u_j$. So, $\mathbf{C}$ is feasible and rational. Hence, $\lambda\vec{A} + (1-\lambda)\vec{B} \in X$. □

Theorem 4.2. *Aggregated Max-min Fairness is achievable.*

Theorem 4.2 implies that the job-wise allocation vector satisfying *AMF* is unique. Note, however, that the allocation matrix satisfying *AMF* may not be unique, as different allocation matrices can produce the same job-wise allocation vector.

4.4 Properties of Aggregate Max-min Fairness

Now, we study whether *AMF* satisfies the properties of Pareto efficiency, envy-freeness, strategy-proofness and sharing incentive.

4.4.1 Pareto Efficiency

Pareto efficiency means that it is not possible to increase the allocation of a job without decreasing the allocation of another job. In our context of distributed job execution, the aggregate amount of resources received by a job J_i from all the sites is given by $\sum_{j=1}^{m} a_{ij}$. To prove that an allocation satisfies Pareto efficiency, we need to show that $\sum_{j=1}^{m} a_{ij}$ cannot be increased without decreasing the aggregate allocation $\sum_{j=1}^{m} a_{kj}$ of at least one other job J_k.

Theorem 4.3. *AMF is Pareto efficient.*

Proof. The job-wise allocation vector $\vec{A}$ of an *AMF* allocation $\mathbf{A}_{n\times m}$ is max-min fair. By the definition of max-min fairness, for any feasible and rational allocation $\mathbf{B}_{n\times m}$ other than $\mathbf{A}_{n\times m}$, if a job J_i satisfies $\sum_{j=1}^{m} b_{ij} > \sum_{j=1}^{m} a_{ij}$, there must be another job J_k such that $\sum_{j=1}^{m} b_{kj} < \sum_{j=1}^{m} a_{kj}$. Therefore, *AMF* is Pareto efficient. □

4.4.2 Envy Freeness

Envy-freeness means that a job would not prefer the resource allocation of any other job. In our context of distributed job execution, a job J_i would envy the allocation of another job J_k if J_i can acquire a larger aggregate amount of useful resources from all the sites using J_k's allocation. Given an allocation $\mathbf{A}_{n\times m}$, if job J_i gets job J_k's allocation, the actual amount of resources J_i can use at each site S_j is $\min\{a_{kj}, d_{ij}\}$ since its demand at site S_j is d_{ij}. Thus, the aggregate amount of resources it can use across all sites is $\sum_{j=1}^{m} \min\{a_{kj}, d_{ij}\}$. Therefore, an allocation $\mathbf{A}_{n\times m}$ satisfies envy-freeness if for any two jobs J_i and J_k, it holds that

$$\sum_{j=1}^{m} \min\{a_{kj}, d_{ij}\} \leq \sum_{j=1}^{m} a_{ij}.$$

Theorem 4.4. *AMF is envy-free.*

Proof. We prove it by contradiction. Assume on the contrary that an *AMF* allocation $\mathbf{A}_{n\times m}$ is not envy-free, *i.e.*, there exist two jobs J_i and J_k satisfying $\sum_{j=1}^{m} \min\{a_{kj}, d_{ij}\} > \sum_{j=1}^{m} a_{ij}$. This implies two facts:

First, the aggregate amount of resources received by job J_k is greater than that received by job J_i: $\sum_{j=1}^{m} a_{kj} > \sum_{j=1}^{m} a_{ij}$.

Second, there exists at least one site S_{j^*} such that $\min\{a_{kj^*}, d_{ij^*}\} > a_{ij^*}$. This indicates $d_{ij^*} > a_{ij^*}$. Thus, if we increase a_{ij^*} and meanwhile decrease a_{kj^*} to maintain the capacity constraint, the actual amount of resources that job J_i can use is increased while that job J_k can use is decreased. Together with the first fact, we can infer that in $\mathbf{A}_{n\times m}$'s job-wise allocation vector $\vec{A}$, it is possible to increase the i-th component by decreasing the k-th component which is larger. This contradicts that $\vec{A}$ is max-min fair. □

4.4.3 Strategy Proofness

In our context of distributed job execution, strategy-proofness means that a job cannot increase its allocation by lying about its demands at *any* set of sites. Suppose a job J_l lies about its demands. Let $\mathbf{A}_{n\times m}$ denote the allocation matrix when all jobs report their true demands and let $\mathbf{A'_{n\times m}}$ denote the allocation matrix when job J_l mis-reports

its demands. For each site $S_j \in \mathcal{M}$, the actual amount of resources J_l can use by lying is $\min\{a'_{lj}, d_{lj}\}$ since the allocation exceeding J_l's demand is useless. Similarly, for any other (honest) job $J_k \neq J_l$, the useful allocation of J_k at site S_j when job J_l lies is $\min\{a'_{kj}, d_{kj}\}$. To prove that an allocation policy satisfies strategy-proofness, we need to show $\sum_{j=1}^{m} \min\{a'_{lj}, d_{lj}\} \leq \sum_{j=1}^{m} a_{lj}$.

To prove that *AMF* is strategy-proof, we have two preparation steps. First, we prove that when a job lies, the total useful allocation of all the jobs cannot increase (Lemma 4.3 and Corollary 4.1). Second, starting from a job that benefits in its allocation, we explore how to identify another job that also benefits in its allocation (Lemma 4.4).

Lemma 4.3. *When a job J_l lies, the total useful allocation of all the jobs at any site cannot increase, i.e., $\forall j \in \mathcal{M}$, $\sum_{i=1}^{n} \min\{a'_{ij}, d_{ij}\} \leq \sum_{i=1}^{n} a_{ij}$.*

Proof. Assume on the contrary that when J_l lies, there exists a site S_{j^*} such that

$$\sum_{i=1}^{n} \min\{a'_{ij^*}, d_{ij^*}\} > \sum_{i=1}^{n} a_{ij^*}.$$

It follows that $\sum_{i=1}^{n} d_{ij^*} > \sum_{i=1}^{n} a_{ij^*}$ Thus, by the Pareto efficiency, we know that the resources at site S_{j^*} must be fully allocated when J_l reports its true demands, *i.e.*, $\sum_{i=1}^{n} a_{ij^*} = u_{j^*}$, where u_{j^*} is the capacity of site S_{j^*}. By the capacity constraint (4.1), we have $u_{j^*} \geq \sum_{i=1}^{n} a'_{ij}$. Putting these relations together,

$$\sum_{i=1}^{n} \min\{a'_{ij^*}, d_{ij^*}\} > \sum_{i=1}^{n} a_{ij^*} = u_{j^*} \geq \sum_{i=1}^{n} a'_{ij^*}.$$

Hence, a contradiction is identified. □

Lemma 4.3 implies the following corollary.

Corollary 4.1. *When a job lies, the total useful allocation over all the sites cannot increase, i.e.,*

$$\sum_{j=1}^{m} \sum_{i=1}^{n} \min\{a'_{ij}, d_{ij}\} \leq \sum_{j=1}^{m} \sum_{i=1}^{n} a_{ij}.$$

Note that when job J_l lies, the aggregate useful allocation of each job J_i is $\sum_{j=1}^{m} \min\{a'_{ij}, d_{ij}\}$. To simplify presentation, we say that a job J_i *benefits* if its aggregate useful allocation increases compared with the scenario that no job lies, *i.e.*, $\sum_{j=1}^{m} \min\{a'_{ij}, d_{ij}\} > \sum_{j=1}^{m} a_{ij}$; and J_i *loses* if its aggregate useful allocation decreases, *i.e.*, $\sum_{j=1}^{m} \min\{a'_{ij}, d_{ij}\} < \sum_{j=1}^{m} a_{ij}$. Similarly, we say that a job J_i *benefits at a site* S_j if its useful allocation at site S_j increases: $\min\{a'_{ij}, d_{ij}\} > a_{ij}$; and J_i *loses at a site* S_j if its useful allocation at site S_j decreases: $\min\{a'_{ij}, d_{ij}\} < a_{ij}$. Obviously, if a job J_i benefits, it must benefit at at least one site.

Lemma 4.4. *Suppose a job J_h benefits and it benefits at a site S_{j^*}. If an honest job J_k loses at site S_{j^*}, then J_k must benefit.*

Proof. Since J_h benefits at site S_{j^*}, we have $\min\{a'_{hj^*}, d_{hj^*}\} > a_{hj^*}$. Thus, in the allocation $\mathbf{A}$, we can increase a_{hj^*} and hence J_h's aggregate allocation by decreasing a_{kj^*} and hence J_k's aggregate allocation. Since $\mathbf{A}$ is an *AMF* allocation, its job-wise allocation vector is max-min fair. Thus, it must hold that $\sum_{j=1}^{m} a_{hj} \geq \sum_{j=1}^{m} a_{kj}$.

On the other hand, if an honest job J_k loses at site S_{j^*}, we have $\min\{a'_{kj^*}, d_{kj^*}\} < a_{kj*}$. Since $a_{kj^*} \leq d_{kj^*}$ due to the demand constraint (4.2), it follows that $a'_{kj^*} < a_{kj^*}$. Thus, in the allocation $\mathbf{A}'$, we can increase a'_{kj^*} and hence J_k's aggregate allocation by decreasing a'_{hj^*} and hence J_h's aggregate allocation. Similarly, since $\mathbf{A}'$ is also an *AMF* allocation, it must hold that $\sum_{j=1}^{m} a'_{kj} \geq \sum_{j=1}^{m} a'_{hj}$.

Since J_h benefits, we have $\sum_{j=1}^{m} \min\{a'_{hj}, d_{hj}\} > \sum_{j=1}^{m} a_{hj}$. Since J_k is honest, we have $\sum_{j=1}^{m} \min\{a'_{kj}, d_{kj}\} = \sum_{j=1}^{m} a'_{kj}$, due to the demand constraints (4.2). Combining the above results, we obtain $\sum_{j=1}^{m} \min\{a'_{kj}, d_{kj}\} = \sum_{j=1}^{m} a'_{kj} \geq \sum_{j=1}^{m} a'_{hj} \geq \sum_{j=1}^{m} \min\{a'_{hj}, d_{hj}\} > \sum_{j=1}^{m} a_{hj} \geq \sum_{j=1}^{m} a_{kj}$. Therefore, J_k benefits. □

Theorem 4.5. *AMF is strategy-proof.*

Proof. We prove it by contradiction. Assume on the contrary that a job J_l benefits by lying about its demands. According to whether a job benefits or not, we divide all the jobs into three sets $\mathcal{J}_{benf}$, $\mathcal{J}_{lose}$ and $\mathcal{J}_{equ}$ (see Figure 4.1 for an illustration). $\mathcal{J}_{benf}$ includes all the jobs that benefit when J_l lies. $\mathcal{J}_{lose}$ includes all the jobs that lose when J_l lies. $\mathcal{J}_{equ}$ includes all the jobs whose aggregate useful allocations remain unchanged,

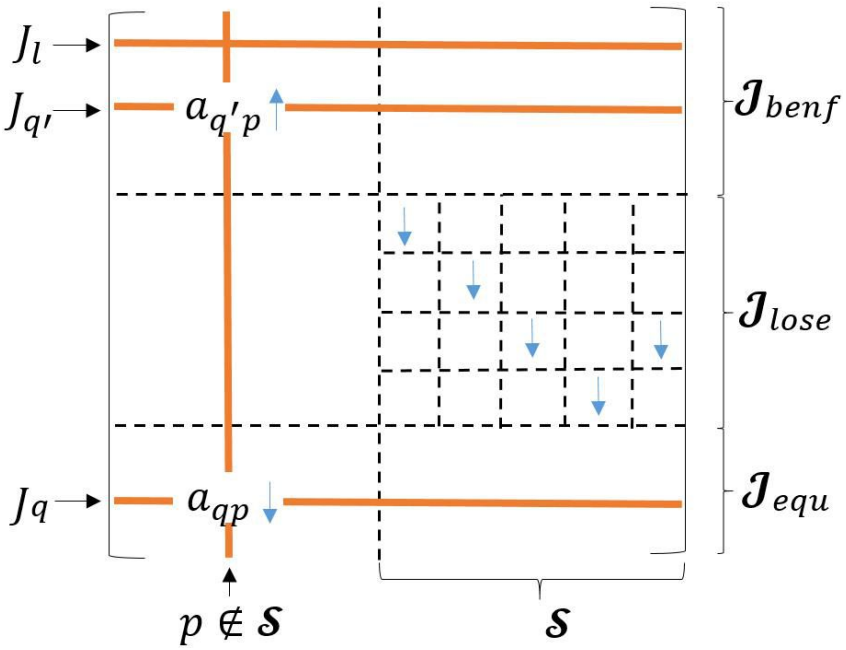


Figure 4.1: Division of jobs and sites (the upward arrow means the allocation of a job at a site increases, and the downward arrow means the allocation of a job at a site decreases)

i.e., $\forall J_i \in \mathcal{J}_{equ}$, $\sum_{j=1}^{m} a'_{ij} = \sum_{j=1}^{m} a_{ij}$. Note that by hypothesis, $J_l \in \mathcal{J}_{benf}$. As a result, all the jobs in $\mathcal{J}_{lose}$ and $\mathcal{J}_{equ}$ are honest. Since $\mathcal{J}_{benf}$ is not empty, $\mathcal{J}_{lose}$ is not empty either (Corollary 4.1). In the following, we show that there must exist a job $J_q \in \mathcal{J}_{equ}$ which benefits, thereby leading to a contradiction.

Let us examine the jobs in $\mathcal{J}_{lose}$. Each of them must lose at at least one site. Define $\mathcal{S} \subseteq \mathcal{M}$ to be the set of sites at which at least one job in $\mathcal{J}_{lose}$ loses. Formally, $\forall j^* \in \mathcal{S}$, $\exists J_k \in \mathcal{J}_{lose}$ such that $a'_{kj^*} < a_{kj^*}$. We first prove that no job in $\mathcal{J}_{benf}$ can benefit at any site belonging to $\mathcal{S}$. Assume on the contrary that $\exists j^* \in \mathcal{S}$ and $\exists J_h \in \mathcal{J}_{benf}$ such that J_h benefits at site S_{j^*}. By the definition of $\mathcal{S}$, we know there is an honest job J_k from $\mathcal{J}_{lose}$ which loses at site S_{j^*}. Then, by Lemma 4.4, J_k should benefit, which contradicts that $J_k \in \mathcal{J}_{lose}$.

If $\mathcal{S} = \mathcal{M}$, the proof is done immediately, as we can conclude that no job in $\mathcal{J}_{benf}$ can benefit. Now assume $\mathcal{S} \neq \mathcal{M}$. Since no job in $\mathcal{J}_{benf}$ can benefit at any site of $\mathcal{S}$, the jobs in $\mathcal{J}_{benf}$ can only get more useful allocations from the set of sites $\mathcal{M}/\mathcal{S}$:

$$\sum_{j \in \mathcal{M}/\mathcal{S}} \sum_{J_i \in \mathcal{J}_{benf}} \min\{a'_{ij}, d_{ij}\} > \sum_{j \in \mathcal{M}/\mathcal{S}} \sum_{J_i \in \mathcal{J}_{benf}} a_{ij}.$$

Therefore, there must exist a site $S_p \in \mathcal{M}/\mathcal{S}$, such that the set of jobs $\mathcal{J}_{benf}$ as a whole benefit at site S_p:

$$\sum_{J_i \in \mathcal{J}_{benf}} \min\{a'_{ip}, d_{ip}\} > \sum_{J_i \in \mathcal{J}_{benf}} a_{ip}.$$

The above inequality implies two facts. First, there exists a job in $\mathcal{J}_{benf}$ which benefits at S_p. Second, by Lemma 4.3, the set of jobs $\mathcal{J}_{equ}$ and $\mathcal{J}_{lose}$ as a whole must lose at site S_p, *i.e.*,

$$\sum_{J_i \in \mathcal{J}_{equ} \cup \mathcal{J}_{lose}} \min\{a'_{ip}, d_{ip}\} < \sum_{J_i \in \mathcal{J}_{equ} \cup \mathcal{J}_{lose}} a_{ip}.$$

As a result, there exists a job in $\mathcal{J}_{equ} \cup \mathcal{J}_{lose}$ which loses at p. Since $p \notin \mathcal{S}$, no job in $\mathcal{J}_{lose}$ loses at S_p. Thus, there must exist an honest job $J_q \in \mathcal{J}_{equ}$ which loses at S_p. Together with the first fact, the conditions of Lemma 4.4 are satisfied. It follows that J_q should benefit, which contradicts that $J_q \in \mathcal{J}_{equ}$. □

4.4.4 Sharing Incentive

Sharing incentive means that each job should be better off sharing the resources than statically partitioning the resources equally among all the jobs. That is, in our context of distributed job execution, the amount of resources allocated to a job J_i at a site S_j should be equal to its demand d_{ij} if $d_{ij} \leq \frac{u_j}{n}$ and should be at least $\frac{u_j}{n}$ if $d_{ij} > \frac{u_j}{n}$, where u_j is the resource capacity of site S_j. Unfortunately, the vanilla version of *AMF* does not always satisfy the sharing incentive property.

Theorem 4.6. *AMF can violate the sharing incentive property.*

Proof. Suppose the resource capacity of each site is 4 and two jobs (J_1 and J_2) are running at two sites (A and B). The resource demands of the jobs are given by the following matrix:

$$\begin{pmatrix} & A & B \\ J_1 & 2 & 2 \\ J_2 & 0 & 3 \end{pmatrix}.$$

In this example, there is no resource contention at site A. To balance the aggregate amounts of resources received by jobs J_1 and J_2, the resources at site B should be allocated between J_1 and J_2 with a ratio $1:3$. As such, the job-wise allocation vector

achieving *AMF* is $\langle 3, 3\rangle$. There is a unique allocation matrix that can produce this job-wise allocation vector:

$$\begin{pmatrix} & A & B \\ J_1 & 2 & 1 \\ J_2 & 0 & 3 \end{pmatrix}.$$

Note that job J_1's resource demand at site B is 2, which is exactly $\frac{1}{2}$ of site B's resource capacity. Under static partitioning in which J_1 owns $\frac{1}{2}$ of each site, J_1 will receive a resource amount 2 from each of A and B. But in the *AMF* allocation, the amount of resources received by J_1 from site B is 1, which is less than 2. This violates the sharing incentive property. □

To guarantee the sharing incentive property, we can restrict the space of possible allocations. That is, besides being feasible and rational (satisfying the site capacity and job demand constraints), an allocation should also satisfy the following constraints:

$$\forall J_i \in \mathcal{J}, S_j \in \mathcal{M} \text{ where } d_{ij} \leq \frac{u_j}{n}, \qquad a_{ij} = d_{ij}, \tag{4.3}$$

$$\forall J_i \in \mathcal{J}, S_j \in \mathcal{M} \text{ where } d_{ij} > \frac{u_j}{n}, \qquad a_{ij} \geq \frac{u_j}{n}. \tag{4.4}$$

It is easy to show that the set of all the job-wise allocation vectors of the allocations satisfying the above constraint is compact and convex. Thus, it is max-min achievable. We refer to *AMF* enhanced with the above constraint as *SIG-AMF* (Sharing Incentive Guaranteed *AMF*). It can be verified that *SIG-AMF* still satisfies the properties of Pareto efficiency, envy-freeness and strategy-proofness as proved earlier.

4.5 Algorithm and Optimization

4.5.1 Programming Algorithms

We now present algorithms to compute resource allocations satisfying *AMF* and *SIG-AMF* for given sets of jobs and sites. First, let us introduce the Max-min Programming algorithm proposed by Radunovic and Le Boudec [52] which can find the max-min fair vector on any achievable set. The algorithm is organized as an iterative process. The key idea of the algorithm is to fix one component of the max-min fair vector in each iteration. The component fixed in each iteration is the smallest component among all those yet to

be fixed in the max-min fair vector, which is found by maximizing the minimal coordinate through a linear program. When all components of the max-min fair vector are fixed, the algorithm stops and returns the result.

We propose an algorithm named Basic Programming which extends from the Max-min Programming algorithm to compute an *AMF* allocation. It takes a set of jobs and a set of sites as inputs. The output is an *AMF* allocation matrix $\mathbf{A}$. The pseudo codes are provided in Algorithm 5.

In Algorithm 5, S records the set of jobs whose aggregate allocations are not yet determined. Initially, S includes all the jobs (line 3). Line 4 to line 21 shows the iterative process. In each iteration, a linear program (line 5) is first solved to maximize T which represents the minimum aggregate allocation among the jobs in S. There are four constraints in the linear program. First, each job in S should have its aggregate allocation at least T. Second, for each job that is not in S, its aggregate allocation must adhere to what has been decided earlier. The last two are the capacity constraints and demand constraints.

On solving the linear program, T is maximized. Let T_{max} denote the maximum T value. For each job J_l in S that has its aggregate allocation equal to T_{max}, we need to check whether it is still possible to increase its aggregate allocation (lines 7-20). A natural approach is to examine a slight variation of the linear program by changing the constraint "$\geq T$" to "$> T_{max}$" for J_l and changing the constraint "$\geq T$" to "$= T_{max}$" for all the other jobs in S (line 13). If the linear constraints are feasible, J_l can receive a larger allocation than T_{max}. Otherwise, J_l's aggregate allocation will be fixed at T_{max} in the following iterations and J_l is removed from S (lines 14-16). Note that some jobs may be concluded without the need to check the linear constraints. Specifically, if a job has all its demands fulfilled, then its allocation cannot be increased (lines 9-11). Moreover, if a job has a demand not fulfilled at a site and that site still has available capacity, then its allocation can be increased (line 12). We filter out these jobs before checking the linear contraints. When S turns empty, the iterative process completes and the *AMF* allocation matrix $\mathbf{A}$ is returned (line 22). The correctness of Algorithm 5 can be easily confirmed by that of the Max-min Programming algorithm [52].

To compute a *SIG-AMF* allocation, we can add the sharing incentive constraints (4.3) and (4.4) presented in Section 4.4.4 to the linear program of line 5.

Algorithm 5: Basic Programming

Input: a set of jobs $\mathcal{J}$ and a set of sites $\mathcal{M}$;
Output: an *AMF* allocation $\mathbf{A}_{n\times m}$;
$S \leftarrow \{J_1, J_2, ..., J_n\}$;
while $S \neq \emptyset$ **do**
  solve the following linear program:
  maximize T;
  subject to: $\forall J_i \in S$: $\sum_{j=1}^{m} a_{ij} \geq T$;
    $\forall J_i \notin S$: $\sum_{j=1}^{m} a_{ij} = A_i$;
    $\forall S_j \in \mathcal{M}$: $\sum_{i=1}^{n} a_{ij} \leq u_j$;
    $\forall J_i \in \mathcal{J}$, $S_j \in \mathcal{M}$: $0 \leq a_{ij} \leq d_{ij}$;
  Let T_{max} be the maximum T in the above linear program;
  for each $J_l \in S$ **do**
    if $\sum_{j=1}^{m} a_{lj} = T_{max}$ **then**
      if $\sum_{j=1}^{m} d_{lj} = T_{max}$ **then**
        $A_l \leftarrow T_{max}$;
        $S \leftarrow S/\{J_l\}$;
      else if there does not exist any $S_j \in \mathcal{M}$ such that $a_{lj} < d_{lj}$ and $\sum_{i=1}^{n} a_{ij} < u_j$ **then**
        check the feasibility of the linear constraints:
        $\sum_{j=1}^{m} a_{lj} > T_{max}$;
        $\forall J_i \in S/\{J_l\}$: $\sum_{j=1}^{m} a_{ij} = T_{max}$;
        $\forall J_i \notin S$: $\sum_{j=1}^{m} a_{ij} = A_i$;
        $\forall S_j \in \mathcal{M}$: $\sum_{i=1}^{n} a_{ij} \leq u_j$;
        $\forall J_i \in \mathcal{J}$, $S_j \in \mathcal{M}$: $0 \leq a_{ij} \leq d_{ij}$;
        if the above constraints are infeasible **then**
          $A_l \leftarrow T_{max}$;
          $S \leftarrow S/\{J_l\}$;
        end if
      end if
    end if
  end for
end while
return $\mathbf{A}_{n\times m}$;

Algorithm 6: Optimization of Estimated Job Completion Times

Input: a sequence of all jobs $\mathcal{J}_{seq}$ and a job-wise allocation vector $\vec{A}$;
Output: an optimized allocation $\mathbf{A}'_{n\times m}$;
while $\mathcal{J}_{seq} \neq \emptyset$ **do**
  $J_k \leftarrow$ the first job in $\mathcal{J}_{seq}$;
  solve the following linear program:
  maximize y_k;
  subject to: $\forall S_j \in \mathcal{M}$: $\sum_{i=1}^{n} a_{ij} \leq u_j$;
    $\forall J_i \notin \mathcal{J}_{seq}$, $S_j \in \mathcal{M}$: $a_{ij} = a'_{ij}$;
    $\forall J_i \in \mathcal{J}_{seq}$, $S_j \in \mathcal{M}$: $0 \leq a_{ij} \leq d_{ij}$;
    $\forall J_i \in \mathcal{J}_{seq}$: $\sum_{j=1}^{m} a_{ij} = A_i$;
    $\forall S_j \in \mathcal{M}$: $a_{kj} = d_{kj} \cdot y_k + z_{kj}$;
    $y_k \geq 0$;
    $\forall S_j \in \mathcal{M}$: $z_{kj} \geq 0$;
  for each $S_j \in \mathcal{M}$ **do**
    $a'_{kj} \leftarrow a_{kj}$;
  end for
  remove J_k from $\mathcal{J}_{seq}$;
end while
return $\mathbf{A}'_{n\times m}$;

4.5.2 Optimization of Job Completion Times

Recall that given a set of jobs and a set of sites, the *AMF* allocation may not be unique, while the job-wise allocation vector satisfying AMF is unique. Algorithm 5 computes only one possible *AMF* allocation. From the *AMF* allocation computed by Algorithm 5, we can derive the job-wise allocation vector satisfying *AMF*. Then, we can take the job-wise allocation vector as a constraint to look for alternative *AMF* allocations with additional considerations. In this section, we propose a heuristic algorithm to find a better *AMF* allocation in terms of estimated job completion times.

The main idea of our heuristic is to use linear programming to sequentially optimize each job's estimated completion time. The pseudo codes are provided in Algorithm 6. The inputs to the algorithm include $\mathcal{J}_{seq}$ and $\vec{A}$. $\mathcal{J}_{seq}$ is a sequence of all the jobs, which specifies the order of optimization. $\vec{A} = \langle A_1, A_2, ..., A_n \rangle$ is the job-wise allocation vector obtained from the *AMF* allocation $\mathbf{A}$ returned by Algorithm 5. It will be used as constraints to ensure that the optimized allocation also meets *AMF*.

Line 3 to line 8 show the optimization procedure. In each round, the first job J_k in $\mathcal{J}_{seq}$ is set as the optimization target (line 4). We first model the completion time of a job. Recall that a job completes only when all of its tasks are finished. A job J_k has d_{kj} tasks to run at each site S_j. If J_k is allocated an amount a_{kj} of resources (computing slots) at site S_j, then the completion time of all the tasks at site S_j is proportional to $\frac{d_{kj}}{a_{kj}}$, where d_{kj} is the remaining number of J_k's tasks at site S_j.[2] Thus, the completion time of job J_k can be estimated as $\max_{j\in\mathcal{M}} \frac{d_{kj}}{a_{kj}}$. To optimize the estimated completion time by linear programming, we need to convert the objective into a linear one. The key step is to rewrite each allocation entry a_{kj} as $d_{kj} \cdot y_k + z_{kj}$, where $y_k \geq 0$ is a job-wise variable and $z_{kj} \geq 0$ is an auxiliary variable. This representation divides a_{kj} into two parts: one part is proportional to d_{kj} which decides the job completion time and the other part z_{kj} fills up the remaining allocation requirement stipulated by A_k to ensure that the resultant allocation meets *AMF*. Then, when y_k is maximized, J_k's estimated completion time is minimized. The logical relations are listed as follows:

$$\min \max_{j\in\mathcal{M}} \frac{d_{kj}}{a_{kj}} \iff \min \max_{j\in\mathcal{M}} \frac{d_{kj}}{d_{kj} \cdot y_k + z_{kj}} \iff \max y_k.$$

Now let us look at the linear program (line 5). The objective is to maximize y_k, hence minimizing J_k's estimated completion time. The first constraint is the capacity constraint. For each job $J_i \notin \mathcal{J}_{seq}$, J_i has been set as the optimization target in a previous round and its allocation is fixed since that round. For each job $J_i \in \mathcal{J}_{seq}$, there are two constraints to be satisfied. The first one is the demand constraint. The second one ensures that the allocation meets *AMF*. For the job J_k to be optimized, there are three additional constraints. The first one is the representation of each a_{ij}. The other two are non-negativity constraints. On solving the linear program, J_i's allocations are settled and will not be changed in any subsequent round (line 6). Then, J_k is removed from $\mathcal{J}_{seq}$ (line 7). The optimization procedure completes after iterating through all the jobs. Finally, the optimized allocation matrix $\mathbf{A}'$ is returned (line 9).

To optimize a *SIG-AMF* allocation, we can add the sharing incentive constraints (4.3) and (4.4) for each job $J_i \in \mathcal{J}_{seq}$ to the linear program of line 5.

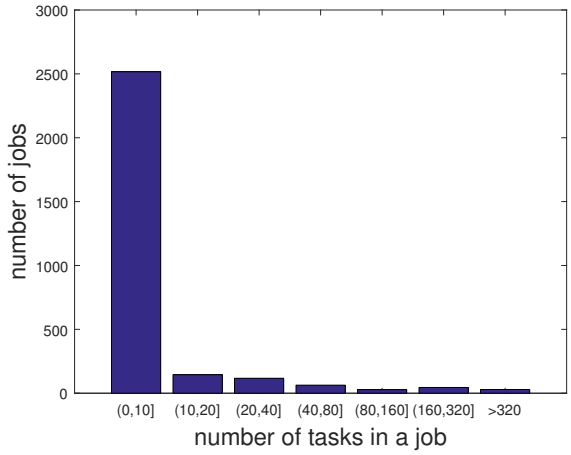
[2]For simplicity, we assume here that all tasks in a job have the same execution time. If estimation on the expected execution times of various tasks is available, the completion time at site S_j can be rewritten as $\frac{w_{kj}}{a_{kj}}$, where w_{kj} is the total execution time of all J_k's tasks at site S_j.

Different strategies can be used to arrange the sequence of jobs J_{seq} for optimization. In this paper, we propose a simple strategy that looks at the total number of tasks in each job across all the sites and sort the jobs in increasing order of their task numbers. In this way, jobs with less tasks are optimized first, since they are more likely to complete early.

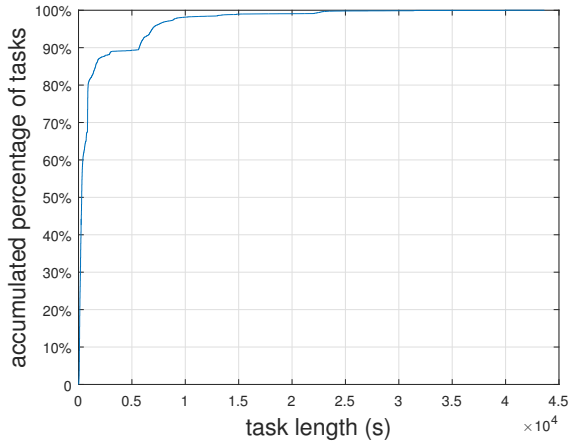
4.5.3 Deployment

Our proposed algorithms can be deployed at a global scheduler of the system. The global scheduler maintains the information about the resource capacities of sites and the resource demands of jobs, based on which the amount of resources allocated to each job at each site can be computed. The resource allocation for each site can be communicated to the local scheduler of the site which is responsible for launching tasks in the computing slots. Whenever a computing slot becomes available, the local scheduler can pick the job whose current resource usage is furthest below its intended allocation and launch a new task of the job in the slot. Specifically, for each job $J_i \in \mathcal{J}$, let v_{ij} denote the number of computing slots currently occupied by J_i at a site S_j. Then, an available computing slot at site S_j can be used to launch a task of the job with the minimum ratio $\frac{v_{ij}}{a_{ij}}$.

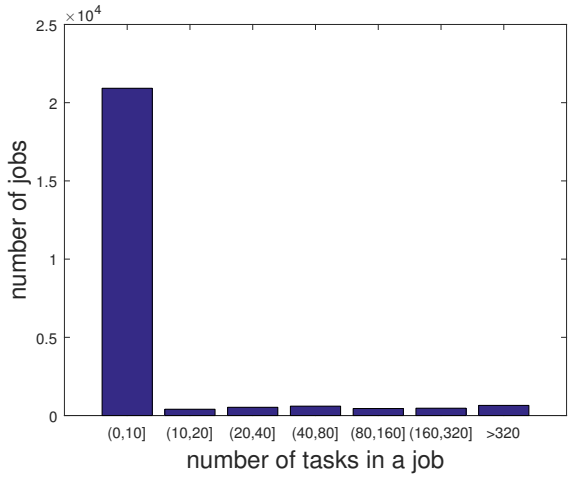
The *AMF* and *SIG-AMF* allocations would change with the resource demands of jobs due to task completions and new job arrivals. When a new job arrives, the allocations need to be recomputed to assign resources to the new job. Similarly, when a running job completes all its tasks at a site (so that its resource demand on that site becomes 0), the allocations can be recomputed among the remaining jobs running at the site. However, the allocations are unlikely to change substantially due to minor updates to the resource demands such as the completion of one task of a job at a site. In particular, when the resource demand of a job at a site exceeds the resource capacity of the site (*i.e.*, $d_{ij} > u_j$), the job-wise allocation vector of the *AMF* allocation is not affected by the actual amount of that job's resource demand at that site (*i.e.*, d_{ij}). Thus, to reduce overheads, the global scheduler can consider adjusting the resource allocation only at instances of new job arrivals and the completion of a job's all tasks at a site. We shall evaluate the impact of such simplification on resource allocation in the experiments.



(a) distribution of task numbers in jobs for Google trace



(b) distribution of task durations for Google trace



(c) distribution of task numbers in jobs for Facebook trace

Figure 4.2: Distribution of task numbers and task lengths

4.6 Experimental Setup

We conduct simulations to compare various allocation policies. This section describes the simulation settings.

Job Characteristics: We use two realistic job traces to drive the simulations: a Google trace and a Facebook trace. The Google trace was collected on a cluster of about 12000 machines over one month at Google [1, 53]. We extract a segment of the trace containing 2944 jobs in a 60-minute window. These jobs include 48504 tasks. Figure 4.2a shows the distribution of task numbers in the jobs. We derive the task lengths from the timestamps of task events recorded in the trace. As shown in Figure 4.2b, the task lengths have a heavy-tailed distribution and have a mean of about 1373.7 seconds. The Facebook trace is the trace FB-2010_samples_24_times_1hr_0.csv from the SWIM workload repository [3, 20], 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 1GB data to process. As a result, there are a total of 1102281 tasks in these jobs. Figure 4.2c shows the distribution of task numbers in the jobs. We generate the task lengths of different jobs according to a Pareto distribution with parameter $\beta = 1.259$ [6] and a mean of 2 seconds. Each task of a job in the Google or Facebook trace is assumed to require one computing slot to execute. We scale the inter-arrival times of the jobs in these traces to simulate different levels of system utilization from 40% to 70%.

We assume that the tasks of a job 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 assigned to the ith site 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 the α value from 0 to 2. When α is set to 0, the expected task distribution is uniform across all sites.

Site Capacity: The default number of sites is set at 10. The resource capacity of each site is set at 20 computing slots.

Resource Allocation Policies: We implement various *AMF* policies. The Basic *AMF* runs Algorithm 5 only to compute the resource allocation. The Optimized *AMF*

runs Algorithm 6 after Algorithm 5 to optimize the job completion times. The Basic *SIG-AMF* and Optimized *SIG-AMF* further guarantee the sharing incentive property. For all the policies, by default, we recompute the resource allocation among active jobs whenever a new job arrives or a running job completes all its tasks at a site. Besides, we also implement a version which further recomputes the resource allocation among active jobs whenever a task of any job is finished at any site. This version is labeled "(Aggressive)" in the policy name. In addition, we also implement the *IMF* policy described at the beginning of Section 4.3 as a baseline.

Performance Metrics: The main objectives of the experiments are to evaluate the fairness in the instantaneous aggregate resource allocations of different jobs under various policies and to study the effectiveness of the optimization proposed in Section 7.2. To do so, we study the following two performance metrics: the standard deviation of jobs' aggregate resource allocations and the average response time of jobs. The aggregate resource allocation of a job is the total number of computing slots occupied by the job over all the sites. We continuously trace the instantaneous aggregate allocations of all the active jobs over time and compute their standard deviation to evaluate the fairness of resource allocation policies. The lower the standard deviation, the more balanced the resource allocation among jobs. It should be noted that due to the discrete setting where the number of computing slots allocated to a job at any time is integer valued and the constraint that a computing slot can be allocated to a new task only after its current task is finished, the standard deviation of the instantaneous aggregate allocations of active jobs is not necessarily equal to that of the allocation matrix output by the programming algorithms. The response time of a job is the duration from its arrival to its completion (*i.e.*, the time when its last task is finished).

4.7 Experimental Results

To facilitate presentation, we shall refer to all the tasks of a job to be executed at a site as a sub-job. First, we compare the aggressive implementation that adjusts the resource allocation whenever a task of any job is finished with the simplified implementation that adjusts the resource allocation only when a sub-job is finished at a site. Figure 4.3 shows

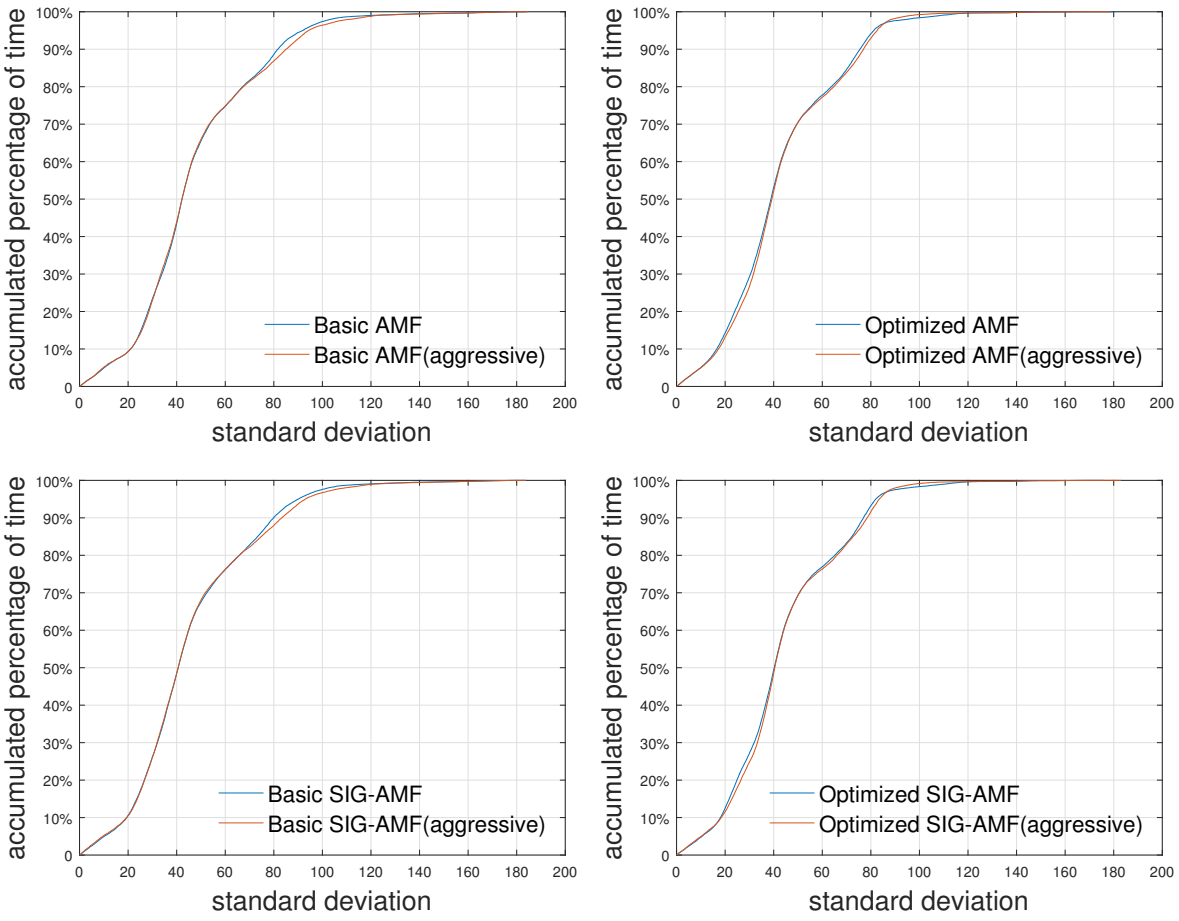


Figure 4.3: Cumulative distribution of standard deviation for Google trace (Zipf = 0, system utilization = 60%

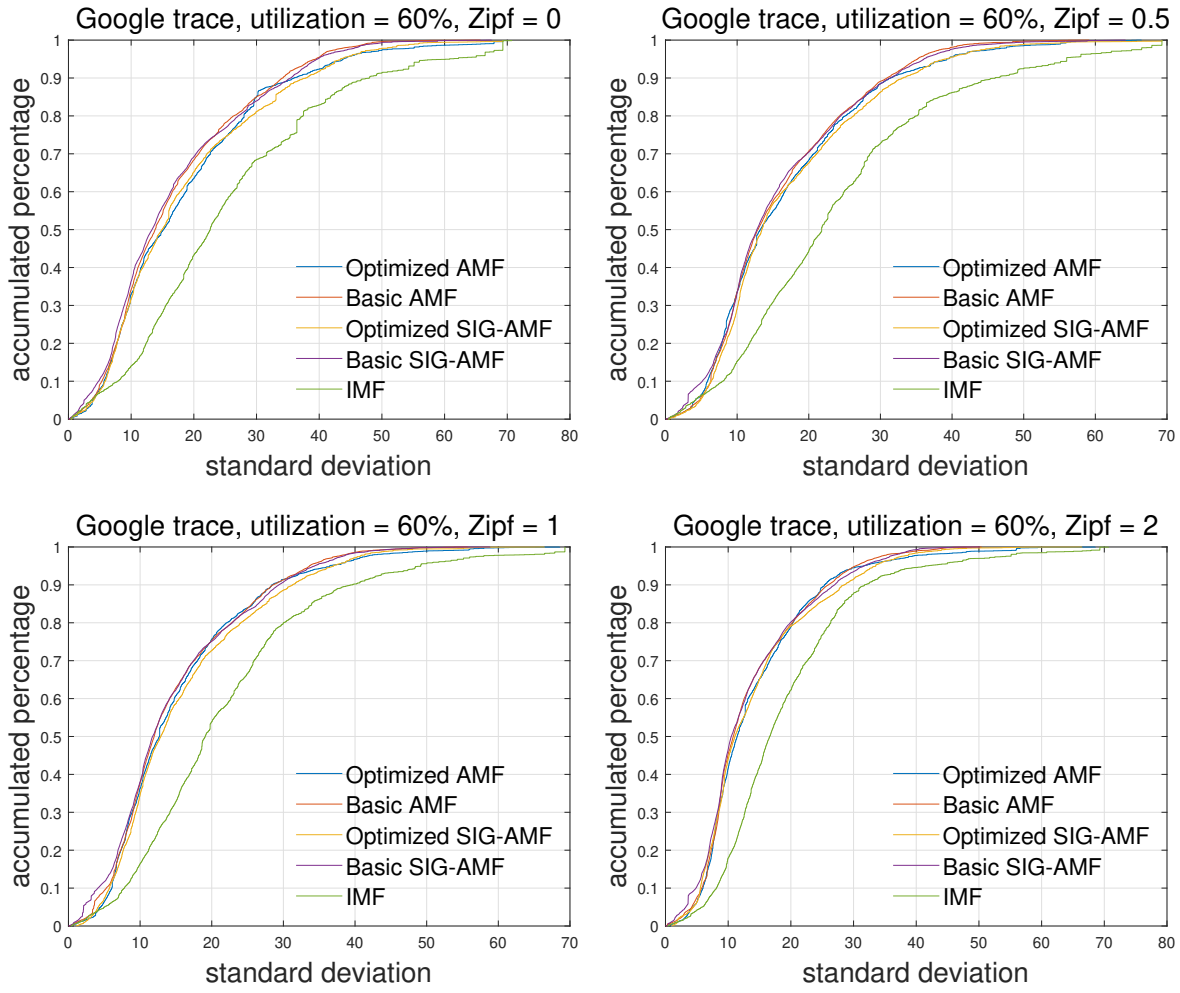


Figure 4.4: Cumulative distribution of standard deviation for Google trace (utilization = 60%)

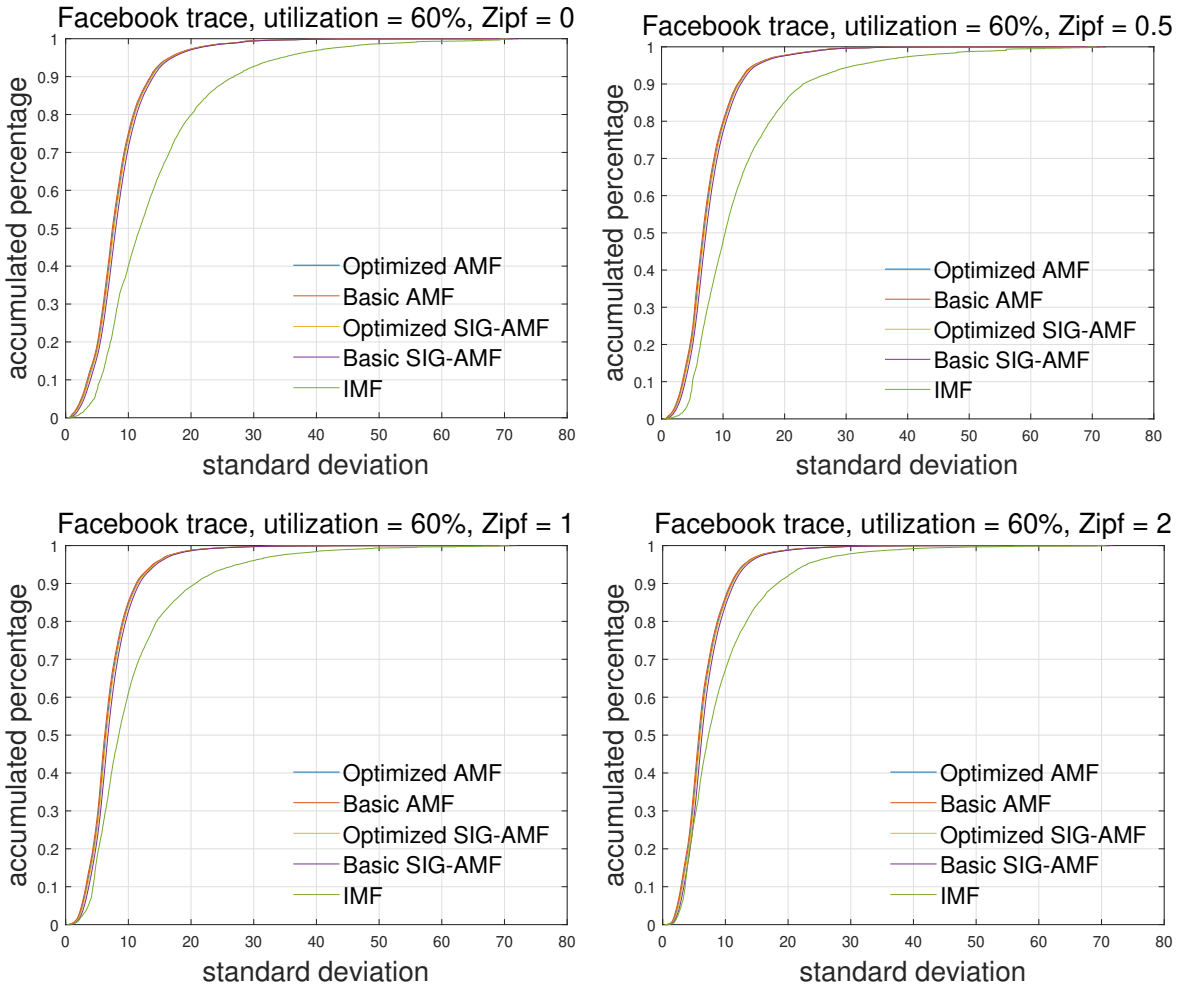


Figure 4.5: Cumulative distribution of standard deviation for Facebook trace (utilization = 60%)

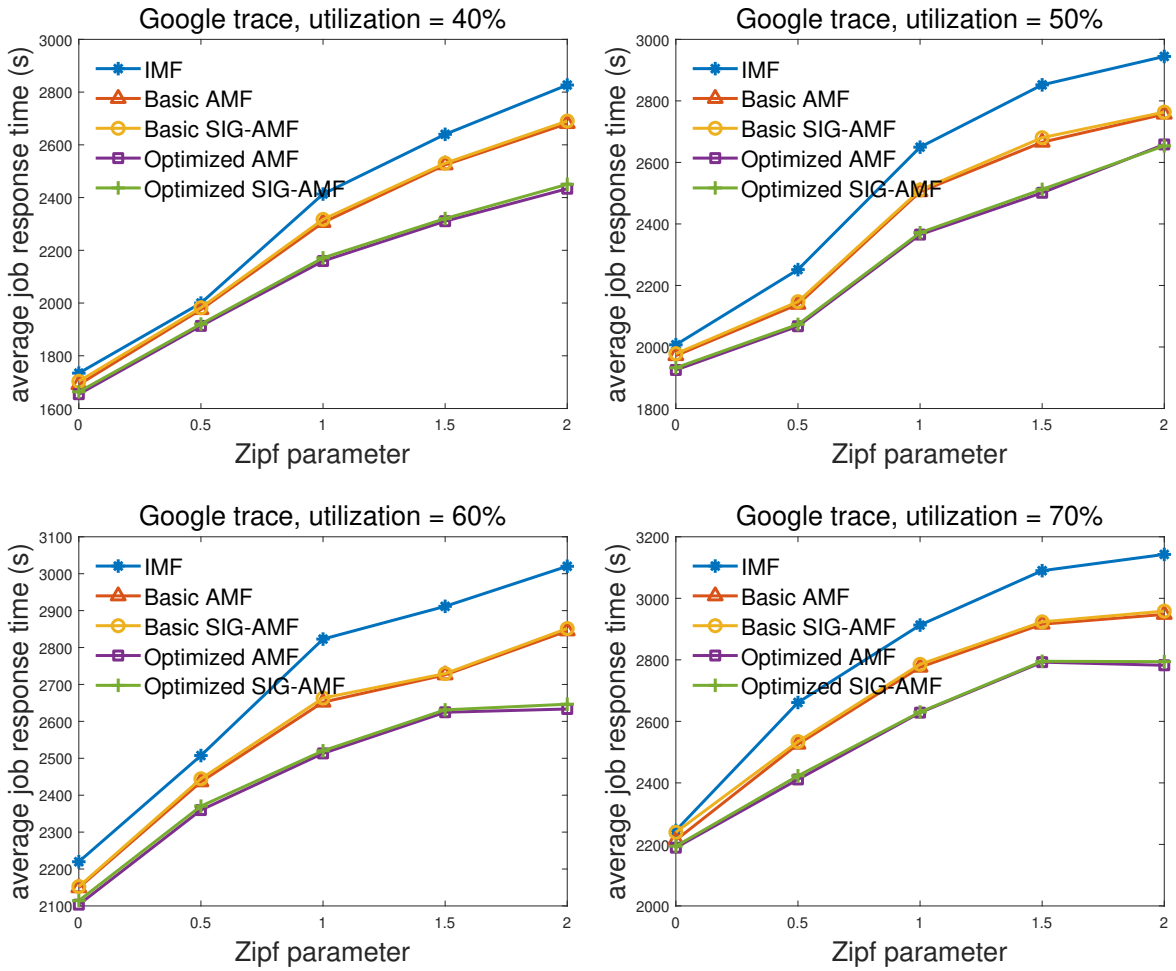


Figure 4.6: Average job response time for Google trace

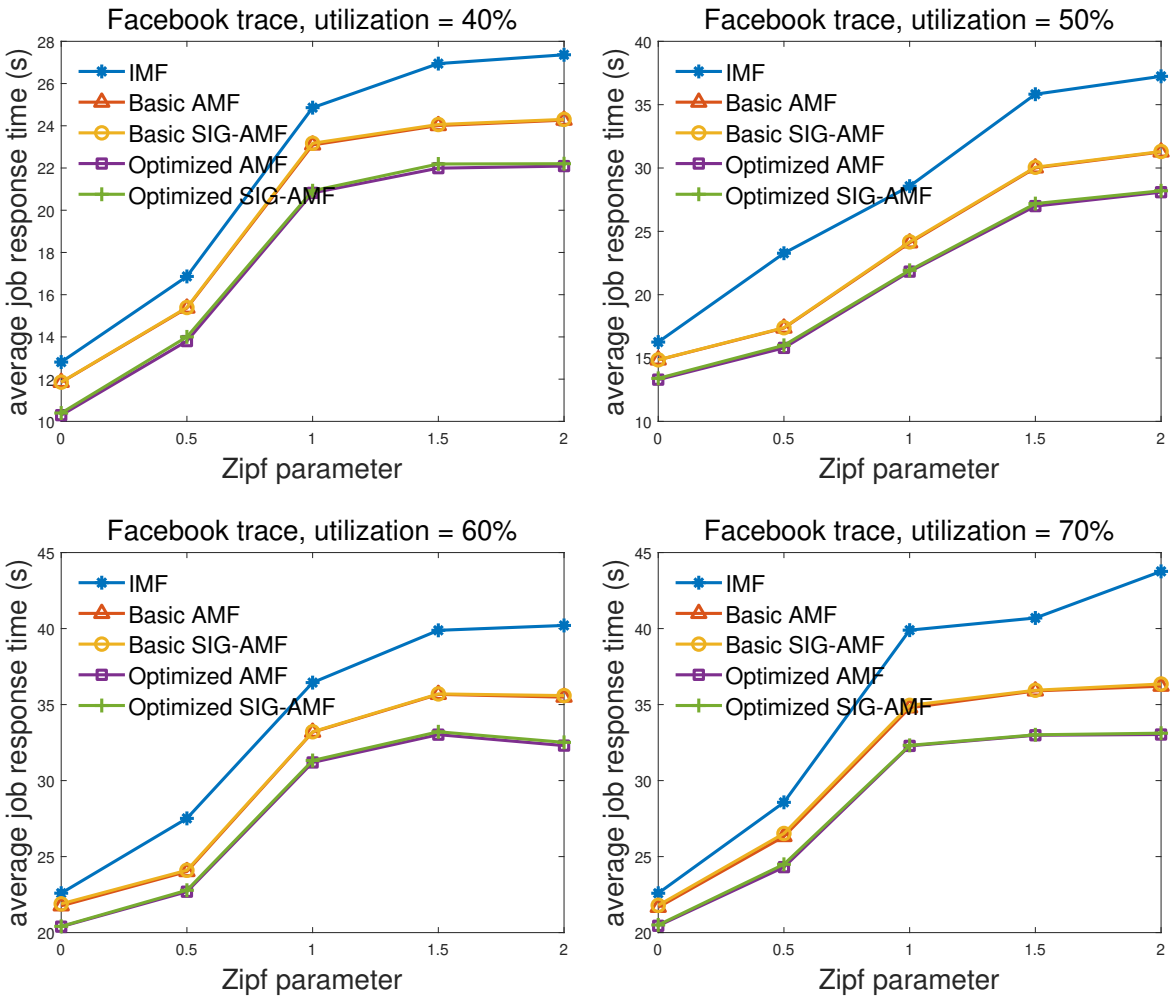


Figure 4.7: Average job response time for Facebook trace

the cumulative distribution of the standard deviation for the aggregate resources received by the active jobs for the Google trace when the Zipf parameter of task distribution is set at 0, the number of sites is set at 30 and the system utilization is 60%. It can be seen that the two implementations have little difference in the standard deviation of resource allocation for all the *AMF* policies. The same trend is consistently observed in the results across all the Zipf parameter values and system utilizations tested as well as for the Facebook trace. This demonstrates that it is not necessary to aggressively recompute the resource allocation at every minor change in the resource demands of jobs. Adjusting the resource allocation only at instances of new job arrivals and sub-job completions is sufficient for balancing the resource allocation. In the rest of this section, we shall focus on comparing the performance of various resource allocation policies under this simplified implementation.

Figures 4.4 and 4.5 show the cumulative distribution of the standard deviation for the aggregate resources received by the active jobs for the Google and Facebook traces at different skewness of task distribution when the system utilization is 60% (the trends for other system utilizations are similar). A point (x, y) on the curve means that the standard deviation is no more than x for y portion of the time. These results confirm that the job-wise aggregate allocations are much more balanced in *AMF* than in *IMF*, even if the task distribution of jobs is near uniform. Various *AMF* policies produce similar standard deviations of resource allocation since all of them ensure that the job-wise aggregate allocation is max-min fair. Since the job-wise allocation vector satisfying max-min fairness is unique, all the basic and optimized *AMF* policies are expected to perform similarly in terms of fairness characterized by the standard deviation. Moreover, in *IMF*, due to independent allocation at each site, the aggregate amount of resources received by a job is generally proportional to its number of sub-jobs. As sub-jobs are completed with the passing of time, the number of unfinished sub-jobs decreases and so does the aggregate resource allocation to the job. Thus, the aggregate amount of resources received by a job can be very unbalanced over its lifetime. In contrast, *AMF* constantly strives for balanced aggregate allocations among all the active jobs. Hence, the aggregate amount of resources received by a job is more consistent throughout its lifetime. This helps to reduce the job response time. Figures 4.6 and 4.7 show the average

job response time for the Google and Facebook traces with different system utilization and skewness of task distribution. As can be seen, all the variants of *AMF* outperform *IMF* in average job response time. The improvement is generally more significant with increasing skewness of task distribution.

Basic *AMF* only ensures that the job-wise allocation vector is max-min fair. In an arbitrary allocation output by Algorithm 5, the amount of resources received by a job from different sites can be quite unbalanced. As a result, some sub-jobs can be finished much faster than other sub-jobs in the same job, leading to longer job response time. By refining the site-level allocation with Algorithm 6, Optimized *AMF* allocates more resources to larger sub-jobs and further improves the job response time over Basic *AMF* as shown in Figures 4.6 and 4.7. As the Zipf parameter increases, the task distribution of a job becomes more skewed so that there is higher potential for Optimized *AMF* to optimize the job response time. Thus, the improvement of Optimized *AMF* over Basic *AMF* generally becomes more significant with increasing Zipf skew parameter.

Compared with *AMF*, *SIG-AMF* produces almost the same the standard deviation of resource allocation and the same average job response time for the traces tested as shown in Figures 4.4 to 4.7. This implies that most of the resource allocations generated by *AMF* satisfy the sharing incentive property. As a result, enforcing the sharing incentive property in *AMF* does not change the resource allocations much.

4.8 Summary

In this chapter, we have studied several resource allocation policies for distributed job execution. We prove that *AMF* satisfies widely recognized properties of Pareto efficiency, envy-freeness and strategy-proofness for fair resource allocation, but the vanilla version of *AMF* does not satisfy the sharing incentive property. We propose an enhanced version of *AMF* to guarantee the sharing incentive property. We present algorithms to implement *AMF* as well as to optimize the job response times under *AMF*. Experiments using real job traces show that *AMF* can significantly reduce unbalanced resource allocation and improve average job response time compared to *IMF* which conducts max-min fair allocation at each site separately, and the improvements are more substantial when the workload distribution of jobs among sites is more skewed.

Chapter 5

Generalized Max-min Fairness for Jobs with Tasks Executable at Multiple Sites

Transferring a large amount of data across different sites (machine clusters or datacenters) can lead to enormous bandwidth costs and network congestion in distributed systems. Thus, it is important to run computing tasks close to their input data so as to reduce data movement. In distributed systems, a distributed file system (such as Hadoop Distributed File System) is often used that comes with some built-in replication strategy. Replication helps make the data files resilient to failures. In addition, it also provides more opportunities for computing tasks to be processed with data locality. With replication, the data to be processed by each task is possibly available at multiple sites, so the task can be assigned to any one of these sites for execution. This gives an additional dimension of freedom in resource allocation, which potentially enables more balanced resource allocation. Meanwhile, the additional freedom also presents a significant challenge to the design and analysis of fair resource allocation policies. In this chapter, we extend the aggregate max-min fairness policy to account for possible replication of job data inputs.

5.1 System Model

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

There is a set of n jobs $J = \{J_1, J_2, ..., J_n\}$ running in the system. Each job to execute in the system includes a set of tasks that process different data partitions and can run in parallel. Same as the model in Chapter 3, the data partitions may be replicated across multiple sites. Each task can be executed at any of the sites where the data partition to be processed is available. These sites are referred to as the *available sites* of the task. 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. All the tasks that have the same available sites compose a *task group*. Suppose there are a total of g task groups $G = \{G_1, G_2, ..., G_g\}$, where $G_k (1 \leq k \leq g)$ represents the set of available sites of the tasks in the k-th task group. A job J_i's resource demand for a task group G_k is the total demand of its tasks in G_k that are not yet executed. If each task occupies one computing slot during execution, then J_i's resource demand for G_k is simply its remaining number of tasks to execute in G_k. We can model the resource demands of all jobs as a matrix $D_{n\times g}$, in which each entry d_{ik} represents job J_i's resource demand for task group G_k. If J_i has no task to run in task group G_k, we define $d_{ik} = 0$. Resource allocation among the jobs can be represented by a three-dimensional $n \times m \times g$ matrix $A_{n\times m\times g}$. Each entry a_{ijk} in $A_{n\times m\times g}$ represents the amount of resources allocated to job J_i by site S_j to run tasks in task group G_k. We refer to $A_{n\times m\times g}$ as an allocation matrix or an allocation for short. Then, any *feasible* allocation must satisfy the site capacity constraints:

$$\forall S_j \in \mathcal{M}, \quad \sum_{i=1}^{n} \sum_{k=1}^{g} a_{ijk} \leq u_j, \tag{5.1}$$

which means that the total amount of resources allocated by each site S_j cannot exceed its capacity. On the other hand, it does not make any sense to allocate more resources than what a job needs for a task group at its available sites. Thus, any *rational* allocation should satisfy the group demand constraints:

$$\forall J_i \in \mathcal{J}, G_k \in \mathcal{G}, \quad 0 \leq \sum_{j:S_j \in G_k} a_{ijk} \leq d_{ik}, \tag{5.2}$$

$$\forall J_i \in \mathcal{J}, G_k \in \mathcal{G}, S_j \notin G_k, \quad a_{ijk} = 0. \tag{5.3}$$

By (5.3), the site capacity constraints can be rewritten as

$$\forall S_j \in \mathcal{M}, \quad \sum_{i=1}^{n} \sum_{k:S_j \in G_k} a_{ijk} \leq u_j. \tag{5.4}$$

We focus on feasible and rational allocations only and aim to achieve the fairness of resource allocation among the jobs.

5.2 Generalized Aggregate Max-min Fairness

We extend the Aggregate Max-min Fairness (*AMF*) policy to deal with jobs that have tasks executable at multiple sites and term it the Generalized Aggregate Max-min Fairness (*GAMF*) policy. *GAMF* also considers all the sites as a united resource pool and defines fairness according to the total amount of resources received by each job from all the sites for all the task groups. The rationale is similar to *AMF* where the system-wide processing rate of each job is decided by its aggregate resource allocation in distributed job execution. Given a three-dimensional allocation matrix $A_{m\times n\times g}$, we define a job-wise allocation vector $\vec{A}$ to represent the aggregate amount of resources received by each job from all the sites for all the task groups:

$$\vec{A} = \langle \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{1jk}, \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{2jk}, ..., \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{njk} \rangle.$$

GAMF requires that the vector $\vec{A}$ is max-min fair. Formally, let $\mathcal{X}$ denote the set of all the feasible and rational allocation matrices. Let X be the set including all the job-wise allocation vectors of the allocation matrices in $\mathcal{X}$.

Definition 5.1. *An allocation $A_{n\times m\times g}$ satisfies Generalized Aggregate Max-min Fairness, if and only if, its job-wise allocation vector $\vec{A}$ is the max-min fair vector over the set X.*

According to Theorem 4.1 in Chapter 4, if the set X is compact and convex, then *GAMF* is achievable.

Lemma 5.1. *The set X is compact.*

Proof. Consider any job-wise allocation vector

$$\vec{A} = \langle \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{1jk}, \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{2jk}, ..., \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{njk} \rangle$$

produced by a feasible and rational allocation matrix $A_{n\times m\times g}$. According to the demand and capacity constraints (*i.e.,* (5.2), (5.3) and (5.4)), for each job J_i, we have

$$0 \leq \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk} = \sum_{j=1}^{m} \sum_{k:S_j \in G_k} a_{ijk} \leq \min\{\sum_{j=1}^{m} u_j, \sum_{k=1}^{g} d_{ik}\}.$$

Thus, the set X is closed and bounded. Therefore, it is compact. □

Lemma 5.2. *The set X is convex.*

Proof. To prove X is convex, we need to prove that for any two vectors $\vec{A}, \vec{B} \in X$ and any $0 \leq \lambda \leq 1$, it holds that $\lambda\vec{A} + (1-\lambda)\vec{B} \in X$.

Suppose that $A_{n\times m\times g}$ and $B_{n\times m\times g}$ are two allocation matrixes. Their corresponding job-wise vectors are $\vec{A}$ and $\vec{B}$ respectively. Consider the allocation $C_{n\times m\times g} = \lambda A_{n\times m\times g} + (1-\lambda)B_{n\times m\times g}$. It is obvious that the corresponding job-wise vector of $C_{n\times m\times g}$ is $\lambda\vec{A} + (1-\lambda)\vec{B}$.

Since $A_{n\times m\times g}$ and $B_{n\times m\times g}$ are feasible, according to the constraints (5.4), for each site S_j, we have

$$0 \leq \lambda \sum_{i=1}^{n} \sum_{k:S_j \in G_k} a_{ijk} \leq \lambda u_j,$$

$$0 \leq (1-\lambda) \sum_{i=1}^{n} \sum_{k:S_j \in G_k} b_{ijk} \leq (1-\lambda) u_j.$$

Then, we can get that

$$0 \leq \lambda \sum_{i=1}^{n} \sum_{k:S_j \in G_k} a_{ijk} + (1-\lambda) \sum_{i=1}^{n} \sum_{k:S_j \in G_k} b_{ijk} = \sum_{i=1}^{n} \sum_{k:S_j \in G_k} c_{ijk} \leq u_j.$$

Similarly, since $A_{n\times m\times g}$ and $B_{n\times m\times g}$ are rational, according to constraints (5.2) for each job J_i and each task group G_k, we have

$$0 \leq \lambda \sum_{j:S_j \in G_k} a_{ijk} \leq \lambda d_{ik},$$

$$0 \leq (1-\lambda) \sum_{j:S_j \in G_k} b_{ijk} \leq (1-\lambda) d_{ik}.$$

Then, we can get that

$$0 \leq \lambda \sum_{j:S_j \in G_k} a_{ijk} + (1-\lambda) \sum_{j:S_j \in G_k} b_{ijk} = \sum_{j:S_j \in G_k} (\lambda a_{ijk} + (1-\lambda) b_{ijk}) = \sum_{j:S_j \in G_k} c_{ijk} \leq d_{ik}.$$

In addition, according to constraints (5.3), for each job J_i, each task group G_k and each site $S_j \notin G_k, a_{ijk} = 0, b_{ijk} = 0$. Then, we can get that

$$c_{ijk} = \lambda a_{ijk} + (1-\lambda) b_{ijk} = 0.$$

Therefore, $C_{n \times m \times g}$ is feasible and rational. Hence, $\lambda \vec{A} + (1-\lambda)\vec{B} \in X$, and the set X is convex. □

Lemmas 5.1 and 5.2 imply that *GAMF* is achievable.

Theorem 5.1. *Generalized Aggregate Max-min Fairness is achievable.*

5.3 Properties of Generalized Aggregate Max-min Fairness

Now, we study whether *GAMF* satisfies the properties of Pareto efficiency, envy-freeness, strategy-proofness and sharing incentive. The analysis of Pareto efficiency and envy-freeness is similar to that for *AMF* in Chapter 4.

5.3.1 Pareto Efficiency

Pareto efficiency means that it is not possible to increase the allocation of a job without decreasing the allocation of another job. In our context of distributed job execution with tasks executable at multiple sites, the aggregate amount of resources received by a job J_i from all sites for all task groups is given by $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk}$. To prove that an allocation satisfies Pareto efficiency, we need to show that $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk}$ cannot be increased without decreasing the aggregate allocation $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ljk}$ of at least one other job J_l.

Theorem 5.2. *GAMF satisfies Pareto efficiency.*

Proof. The job-wise allocation vector $\vec{A}$ of a *GAMF* allocation $A_{n\times m\times g}$ is max-min fair. By the definition of max-min fairness, for any feasible and rational allocation $B_{n\times m\times g}$ other than $A_{n\times m\times g}$, if a job J_i satisfies

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} b_{ijk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{ijk},$$

there must be another job J_l such that

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} b_{ljk} < \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{ljk}.$$

Therefore, *GAMF* is Pareto efficient. □

5.3.2 Envy Freeness

Envy-freeness means that a job would not prefer the resources allocation of another job in the system. In our context of distributed job execution with tasks executable at multiple sites, a job J_i would envy the allocation of another job J_l if J_i can acquire a larger aggregate amount of useful resources from all the sites for all the task groups using J_l's allocation. Given an allocation $A_{n\times m\times g}$, if job J_i gets job J_l's allocation, the aggregate amount of resources for each task group G_k is $\sum_{j:S_j\in G_k} a_{a_{ljk}}$. Hence, the aggregate amount of resources it can use across all the task groups is $\sum_{k=1}^{g}\min\{\sum_{j:S_j\in G_k} a_{ljk}, d_{ik}\}$. Therefore, an allocation $A_{n\times m\times g}$ satisfies envy-freeness if for any two jobs J_i and J_l, it holds that

$$\sum_{k=1}^{g}\min\{\sum_{j:S_j\in G_k} a_{ljk}, d_{ik}\} \le \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{ijk}.$$

Theorem 5.3. *GAMF satisfies envy-freeness.*

Proof. We prove it by contradiction. Assume on the contrary that an *GAMF* allocation $A_{n\times m\times g}$ is not envy-free, *i.e.*, there exist two jobs J_i and J_l satisfying

$$\sum_{k=1}^{g}\min\{\sum_{j:S_j\in G_k} a_{ljk}, d_{ik}\} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{ijk}.$$

This implies two facts:

First, the aggregate amount of resources received by job J_l is greater than that received by job J_i:

$$\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ljk} > \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk}.$$

Second, there exists at least one task group G_{k^*} such that

$$\min\{ \sum_{j:S_j \in G_k} a_{ljk^*}, d_{ik^*}\} > \sum_{j:S_j \in G_k} a_{ijk^*}.$$

This indicates that $d_{ik^*} > \sum_{j:S_j \in G_k} a_{ijk^*}$. Thus, if we increase a_{ijk^*} for some site $S_j \in G_k$ and meanwhile decrease a_{ljk^*} to maintain the capacity constraint, the actual amount of resources that job J_i can use is increased while that job J_l can use is decreased. Together with the first fact, we can infer that in $A_{n\times m\times g}$'s job-wise allocation vector $\vec{A}$, it is possible to increase the i-th component $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk}$ by decreasing the l-th component $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ljk}$ which is larger. This contradicts that $\vec{A}$ is max-min fair. □

5.3.3 Strategy Proofness

In our context of distribution job execution with tasks executable at multiple sites, strategy-proofness means that a job cannot increase its allocation by lying about its demand for *any* set of task groups. Suppose a job J_r lies about its demands. Let $A_{n\times m\times g}$ denote the allocation matrix when all jobs report their true demands and let $A'_{n\times m\times g}$ denote the allocation matrix when job J_r mis-reports its demands. For each task group $G_k \in \mathcal{G}$, the actual amount of resources J_r can use by lying is $\min\{\sum_{j:S_j \in G_k} a'_{rjk}, d_{rk}\}$ since the allocation exceeding J_r's demand is useless. Similarly, for any other (honest) job $J_l \neq J_r$, the useful allocation of J_l for task group G_k when job J_r lies is $\min\{\sum_{j:S_j \in G_k} a'_{ljk}, d_{lk}\}$. To prove that an allocation policy satisfies strategy-proofness, we need to show that J_r cannot benefit in its allocation, *i.e.,*

$$\sum_{k=1}^{g} \min\{ \sum_{j:S_j \in G_k} a'_{rjk}, d_{rk}\} \leq \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{rjk}.$$

The proof is more complicated than that for *AMF* in Chapter 4.

Theorem 5.4. *GAMF is strategy-proof.*

Proof. Let $A_{n\times m\times g}$ be the allocation matrix when all jobs report their true demands. Let $A'_{n\times m\times g}$ be the allocation matrix when a job J_r lies about its demands.

We prove the strategy-proofness by contradiction. Assume on the contrary that when a job J_r lies about its demands, its useful allocation increases, i.e.,

$$\sum_{k=1}^{g} \min\left\{\sum_{j:S_j\in G_k} a'_{rjk}, d_{rk}\right\} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{rjk}. \tag{5.5}$$

It follows from (5.5) that

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{rjk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{rjk}. \tag{5.6}$$

First, we construct another allocation matrix $B'_{n\times m\times g}$ based on $A'_{n\times m\times g}$ as follows:

(i) For each job $J_i \neq J_r$, we define $b'_{ijk} = a'_{ijk}$ for any $1 \leq j \leq m$ and $1 \leq k \leq g$. Since J_i reports its true demands, by the group demand constraints (5.2), for each task group G_k, we have $\sum_{j:S_j\in G_k} b'_{ijk} = \sum_{j:S_j\in G_k} a'_{ijk} \leq d_{ik}$.

(ii) For the job J_r and each task group G_k, if $\sum_{j:S_j\in G_k} a'_{rjk} \leq d_{rk}$, we define $b'_{rjk} = a'_{rjk}$ for any $1 \leq j \leq m$. Then, we have $\sum_{j:S_j\in G_k} b'_{rjk} = \sum_{j:S_j\in G_k} a'_{rjk} \leq d_{rk}$.

(iii) Otherwise, if $\sum_{j:S_j\in G_k} a'_{rjk} > d_{rk}$, we reduce a'_{rjk} for some sites S_j where $S_j \in G_k$ to produce b'_{rjk} so that $\sum_{j:S_j\in G_k} b'_{rjk} = d_{rk}$. The particular entries a'_{rjk} to reduce may be chosen arbitrarily.

In a nutshell, $B'_{n\times m\times g}$ reduces some entries a'_{rjk} $(1 \leq j \leq m, 1 \leq k \leq g)$ for job J_r in $A'_{n\times m\times g}$ to record J_r's useful allocation, while keeping all the entries for other jobs unchanged. It is easy to see that the allocation matrix $B'_{n\times m\times g}$ satisfies all the group demand and site capacity constraints ((5.2), (5.3) and (5.4)). Moreover, for each job J_i and each task group G_k, J_i's useful allocation for G_k under $A'_{n\times m\times g}$ is the same as its total allocation for G_k under $B'_{n\times m\times g}$, i.e.,

$$\forall i,k, \quad \min\left\{\sum_{j:S_j\in G_k} a'_{ijk}, d_{ik}\right\} = \sum_{j:S_j\in G_k} b'_{ijk}.$$

Thus, for each job J_i, J_i's useful allocation under $A'_{n\times m\times g}$ is the same as its total allocation under $B'_{n\times m\times g}$, i.e.,

$$\forall i, \quad \sum_{k=1}^{g} \min\left\{\sum_{j:S_j\in G_k} a'_{ijk}, d_{ik}\right\} = \sum_{k=1}^{g}\sum_{j:S_j\in G_k} b'_{ijk}.$$

By (5.5), J_r's total allocation increases from $A_{n\times m\times g}$ to $B'_{n\times m\times g}$, i.e.,

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} b'_{rjk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{rjk}. \tag{5.7}$$

Note that both $A_{n\times m\times g}$ and $B'_{n\times m\times g}$ satisfy all the group demand and site capacity constraints ((5.2), (5.3) and (5.4)). Since $A_{n\times m\times g}$ is a GAMF allocation where the vector representing the total allocation of each job is max-min fair, by the definition of max-min fairness, there must exist another job $J_{i_1} \neq J_r$ such that J_{i_1}'s total allocation decreases from $A_{n\times m\times g}$ to $B'_{n\times m\times g}$ and J_{i_1}'s total allocation under $A_{n\times m\times g}$ is no more than J_r's total allocation under $A_{n\times m\times g}$, i.e.,

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{rjk} \geq \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{i_1jk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} b'_{i_1jk}. \tag{5.8}$$

Since $J_{i_1} \neq J_r$, by the construction of $B'_{n\times m\times g}$, J_{i_1}'s total allocations are the same under $B'_{n\times m\times g}$ and $A'_{n\times m\times g}$, i.e.,

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} b'_{i_1jk} = \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{i_1jk}. \tag{5.9}$$

By (5.9), (5.8) can be rewritten as

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{rjk} \geq \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{i_1jk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{i_1jk}. \tag{5.10}$$

Next, similar to the construction of $B'_{n\times m\times g}$ based on $A'_{n\times m\times g}$, we construct another allocation matrix $B_{n\times m\times g}$ based on $A_{n\times m\times g}$ to record J_r's useful allocation under its mis-reported demands when it lies. Specifically:

(i) For each job $J_i \neq J_r$, we define $b_{ijk} = a_{ijk}$ for any $1 \leq j \leq m$ and $1 \leq k \leq g$. Since J_i reports its true demands, by the group demand constraints (5.2), for each task group G_k, we have $\sum_{j:S_j\in G_k} b_{ijk} = \sum_{j:S_j\in G_k} a_{ijk} \leq d_{ik}$.

(ii) For the job J_r and each task group G_k, if $\sum_{j:S_j\in G_k} a_{rjk} \leq d'_{rk}$ where d'_{rk} is J_r's mis-reported demand when it lies, we define $b_{rjk} = a_{rjk}$ for any $1 \leq j \leq m$. Then, we have $\sum_{j:S_j\in G_k} b_{rjk} = \sum_{j:S_j\in G_k} a_{rjk} \leq d'_{rk}$.

(iii) Otherwise, if $\sum_{j:S_j \in G_k} a_{rjk} > d'_{rk}$, we reduce a_{rjk} for some sites S_j where $S_j \in G_k$ to produce b_{rjk} so that $\sum_{j:S_j \in G_k} b_{rjk} = d'_{rk}$. The particular entries a_{rjk} to reduce may be chosen arbitrarily.

In a nutshell, $B_{n\times m\times g}$ reduces some entries a_{rjk} $(1 \leq j \leq m, 1 \leq k \leq g)$ for job J_r in $A_{n\times m\times g}$ to record J_r's useful allocation under its mis-reported demands when it lies, while keeping all the entries for other jobs unchanged. It follows that the allocation matrix $B_{n\times m\times g}$ satisfies all the group demand and site capacity constraints ((5.2), (5.3) and (5.4)) when J_r lies about its demands. Moreover, for each job J_i and each task group G_k, J_i's useful allocation for G_k under $A_{n\times m\times g}$ is the same as its total allocation for G_k under $B_{n\times m\times g}$, i.e.,

$$\forall i, k, \quad \min\left\{\sum_{j:S_j \in G_k} a_{ijk}, d'_{ik}\right\} = \sum_{j:S_j \in G_k} b_{ijk}.$$

Thus, for each job J_i, J_i's useful allocation under $A_{n\times m\times g}$ is the same as its total allocation under $B_{n\times m\times g}$, i.e.,

$$\forall i, \quad \sum_{k=1}^{g} \min\left\{\sum_{j:S_j \in G_k} a_{ijk}, d'_{ik}\right\} = \sum_{k=1}^{g} \sum_{j:S_j \in G_k} b_{ijk}.$$

Since $J_{i_1} \neq J_r$, by the construction of $B_{n\times m\times g}$, J_{i_1}'s total allocations are the same under $B_{n\times m\times g}$ and $A_{n\times m\times g}$, i.e.,

$$\sum_{k=1}^{g} \sum_{j:S_j \in G_k} b_{i_1jk} = \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{i_1jk}. \tag{5.11}$$

Now, by (5.10) and (5.11), we have

$$\sum_{k=1}^{g} \sum_{j:S_j \in G_k} b_{i_1jk} > \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a'_{i_1jk}, \tag{5.12}$$

i.e., J_{i_1}'s total allocation increases from $A'_{n\times m\times g}$ to $B_{n\times m\times g}$.

Note that both $A'_{n\times m\times g}$ and $B_{n\times m\times g}$ satisfy all the group demand and site capacity constraints ((5.2), (5.3) and (5.4)) when J_r lies about its demands. Since $A'_{n\times m\times g}$ is a GAMF allocation (under the mis-reported demands when J_r lies) where the vector representing the total allocation of each job is max-min fair, by the definition of max-min

fairness, there must exist another job $J_{i_2} \neq J_{i_1}$ such that J_{i_2}'s total allocation decreases from $A'_{n\times m\times g}$ to $B_{n\times m\times g}$ and J_{i_2}'s total allocation under $A'_{n\times m\times g}$ is no more than J_{i_1}'s total allocation under $A'_{n\times m\times g}$, i.e.,

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{i_1jk} \geq \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{i_2jk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} b_{i_2jk}. \tag{5.13}$$

Putting (5.6), (5.10), (5.13) together, we have

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{rjk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{i_2jk},$$

which implies that $J_{i_2} \neq J_r$. Hence, by the construction of $B_{n\times m\times g}$, J_{i_2}'s total allocations are the same under $B_{n\times m\times g}$ and $A_{n\times m\times g}$, i.e.,

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} b_{i_2jk} = \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{i_2jk}. \tag{5.14}$$

By (5.14), (5.13) can be rewritten as

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{i_1jk} \geq \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{i_2jk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{i_2jk}. \tag{5.15}$$

Putting (5.6), (5.10), (5.15) together, we see a sequence of distinct jobs J_r, J_{i_1}, J_{i_2} with non-increasing total allocations under $A_{n\times m\times g}$ (and under $A'_{n\times m\times g}$).

Since $J_{i_2} \neq J_r$, by the construction of $B'_{n\times m\times g}$, J_{i_2}'s total allocations are the same under $B'_{n\times m\times g}$ and $A'_{n\times m\times g}$, i.e.,

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} b'_{i_2jk} = \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a'_{i_2jk}. \tag{5.16}$$

It follows from (5.15) and (5.16) that

$$\sum_{k=1}^{g}\sum_{j:S_j\in G_k} b'_{i_2jk} > \sum_{k=1}^{g}\sum_{j:S_j\in G_k} a_{i_2jk}, \tag{5.17}$$

i.e., J_{i_2}'s total allocation increases from $A_{n\times m\times g}$ to $B'_{n\times m\times g}$.

By similar arguments for the derivation from (5.7) to (5.10), we can find another job $J_{i_3} \neq J_r, J_{i_1}, J_{i_2}$ such that

$$\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{i_2jk} \geq \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{i_3jk} > \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a'_{i_3jk}. \tag{5.18}$$

This then implies

$$\sum_{k=1}^{g} \sum_{j:S_j \in G_k} b_{i_3jk} > \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a'_{i_3jk}, \tag{5.19}$$

i.e., J_{i_3}'s total allocation increases from $A'_{n\times m\times g}$ to $B_{n\times m\times g}$.

By similar arguments for the derivation from (5.12) to (5.15), we can find another job $J_{i_4} \neq J_r, J_{i_1}, J_{i_2}, J_{i_3}$ such that

$$\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a'_{i_3jk} \geq \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a'_{i_4jk} > \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{i_4jk}. \tag{5.20}$$

This then implies

$$\sum_{k=1}^{g} \sum_{j:S_j \in G_k} b'_{i_4jk} > \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{i_4jk}, \tag{5.21}$$

i.e., J_{i_4}'s total allocation increases from $A_{n\times m\times g}$ to $B'_{n\times m\times g}$.

Now, putting (5.6), (5.10), (5.15), (5.18), (5.20) together, we have a longer sequence of distinct jobs J_r, J_{i_1}, J_{i_2}, J_{i_3}, J_{i_4} with non-increasing total allocations under $A_{n\times m\times g}$ (and under $A'_{n\times m\times g}$).

The above process can be repeated over and over to give us an infinite sequence of distinct jobs J_r, J_{i_1}, J_{i_2}, J_{i_3}, J_{i_4}, J_{i_5}, J_{i_6}, ... with non-increasing total allocations under $A_{n\times m\times g}$ and under $A'_{n\times m\times g}$. This contradicts to the fact that there are a finite number of jobs.

Hence, the theorem is proven. □

5.3.4 Sharing Incentive

Sharing incentive means that each job should be better off sharing the resources than statically partitioning the resources equally among all the jobs. We have claimed in Chapter 4 that *AMF* can violate the sharing incentive property (Theorem 4.6), and

AMF can be regarded as a special case of *GAMF* when the numbers of tasks' available sites are all equal to 1. Thus, *GAMF* can also violate the sharing incentive property.

In our context of distributed job execution with tasks executable at multiple sites, we can restrict the space of possible allocations to guarantee the sharing incentive property. For each job, if it owns $1/n$ of the system (i.e. it owns a capacity of u_j/n at each site), we can compute the maximum amount of resources it can actually use simultaneously based on the available sites of its task groups. To do so, we can formulate and solve the following linear program to compute the maximum amount of resources that each job J_i can use.

$$\max \quad \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk} \qquad \text{(LP)}$$

$$\text{s.t.} \quad \forall S_j \in \mathcal{M}, \qquad \sum_{k:S_j \in G_k} a_{ijk} \leq \frac{u_j}{n}, \tag{5.22}$$

$$\forall G_k \in \mathcal{G}, \qquad 0 \leq \sum_{j:S_j \in G_k} a_{ijk} \leq d_{ik}, \tag{5.23}$$

where (5.22) is the site capacity constraint and (5.23) is the group demand constraint.

Let X_i denote the maximum amount of resources that can be used by each job J_i. Then, to assure the sharing incentive property, an allocation should satisfy the following constraints:

$$\forall J_i \in \mathcal{J}, \quad \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk} \geq X_i. \tag{5.24}$$

It can be inferred that the set of all the job-wise allocation vectors of the allocations satisfying the above constraint is compact and convex. Thus, a *GAMF* allocation satisfying sharing incentive is max-min achievable. We refer to *GAMF* enhanced with the above constraint as *SIG-GAMF* (Sharing Incentive Guaranteed *GAMF*). It is easy to prove that *SIG-GAMF* also satisfies the properties of Pareto efficiency, envy-freeness and strategy-proofness following the methodology of previous analysis..

5.4 Algorithms

In this section, we present algorithms for computing resource allocations satisfying *GAMF* and *SIG-GAMF* with given sets of jobs, task groups and sites. Similar to the Basic Pro-

gramming algorithm shown in Chapter 4, we extend the Max-min Programming algorithm [52] to compute a *GAMF* allocation. The algorithm takes as inputs the information of a set of jobs, the task groups of the tasks in the job, the available sites of each task group, and the processing capacity of each site. The output is a three-dimensional *GAMF* allocation matrix $A_{n\times m\times g}$. Algorithm 7 shows the details of the algorithm.

In Algorithm 7, the set S includes all the jobs whose aggregate allocations are not decided. Initially, S contains all the jobs (line 3). The iterations are shown from line 4 to line 24. In each iteration, a linear program is formulated (line 8). The objective of the linear program is to maximize the minimum job-wise aggregate allocation among the jobs in S. Four constraints are involved in the linear program. The first constraint defines T which represents the minimum aggregate allocation among all the jobs in S. The second constraint says that for each job not in S, its aggregate allocation must be consistent to what has been determined earlier. The third constraint is the site capacity constraint. The fourth constraint is the group demand constraint. Let T_{max} be the maximum T value obtained by solving the linear program. Similar to the Basic Programming algorithm in Chapter 4, we check whether it is possible to increase the aggregate allocation of each job $J_l \in S$ whose aggregate allocation equals T_{max} (lines 10-23). If J_l's aggregate allocation cannot be further increased, we fix J_l's aggregate allocation at T_{max} and remove J_l from S (lines 12-14 and lines 17-19). When S becomes empty, the algorithm completes and returns the *GAMF* allocation matrix $A_{n\times m\times g}$. To compute the *SIG-GAMF* allocation, the sharing incentive constraint (5.24) can be added to the linear programs of line 8 and line 16.

We also present a heuristic similar to the optimization for *AMF* in Chapter 4. The *GAMF* allocation might not be unique given a set of jobs, a set of sites and a set of task groups, while the job-wise allocation vector satisfying *GAMF* is unique. Algorithm 7 just computes one possible allocation for *GAMF*. From the *GAMF* allocation computed by Algorithm 7, we can derive the job-wise allocation vector satisfying *GAMF*. Then, we can take the job-wise allocation vector as a constraint to look for alternative *GAMF* allocations with additional considerations. We now propose a heuristic to find a better *GAMF* allocation among all the possible allocations in terms of estimated job completion times.

Algorithm 7: Computation of *GAMF*

Input: a set of jobs $\mathcal{J}$, a set of sites $\mathcal{M}$, a set of task groups $\mathcal{G}$, and the resource demands of jobs for task groups $\mathcal{D}$;
Output: an *GAMF* allocation $\mathbf{A}_{n\times m\times g}$;
$S \leftarrow \{J_1, J_2, ..., J_n\}$;
for each $J_i \in \mathcal{J}, G_k \in \mathcal{G}, S_j \notin G_k$ **do**
  $a_{ijk} \leftarrow 0$
end for
while $S \neq \emptyset$ **do**
  solve the following linear program:
  maximize T;
  subject to: $\forall J_i \in S$: $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk} \geq T$;
  $\forall J_i \notin S$: $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk} = A_i$;
  $\forall S_j \in \mathcal{M}$: $\sum_{i=1}^{n} \sum_{k:S_j \in G_k} a_{ijk} \leq u_j$;
  $\forall J_i \in \mathcal{J}$, $G_k \in \mathcal{G}$, $0 \leq \sum_{j:S_j \in G_k} a_{ijk} \leq d_{ik}$;
  Let T_{max} be the maximum T in the above linear program;
  for each $J_l \in S$ **do**
    if $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ljk} = T_{max}$ **then**
      if $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} d_{ljk} = T_{max}$ **then**
        $A_l \leftarrow T_{max}$;
        $S \leftarrow S/\{J_l\}$;
      else if there does not exist any $S_j \in \mathcal{M}$ such that $\sum_{i=1}^{n} \sum_{k:S_j \in G_k} a_{ijk} < u_j$ and any $G_k \in \mathcal{G}$ such that $\sum_{j:S_j \in G_k} a_{ljk} < d_{lk}$ **then**
        check the feasibility of the linear constraints:
        $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ljk} > T_{max}$;
        $\forall J_i \in S/\{J_l\}$: $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk} = T_{max}$;
        $\forall J_i \notin S$: $\sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk} = A_i$;
        $\forall S_j \in \mathcal{M}$: $\sum_{i=1}^{n} \sum_{k:S_j \in G_k} a_{ijk} \leq u_j$;
        $\forall J_i \in \mathcal{J}$, $G_k \in \mathcal{G}$, $0 \leq \sum_{j:S_j \in G_k} a_{ijk} \leq d_{ik}$;
        if the above constraints are infeasible **then**
          $A_l \leftarrow T_{max}$;
          $S \leftarrow S/\{J_l\}$;
        end if
      end if
    end if
  end for
end while
return $\mathbf{A}_{n\times m\times g}$;

The main idea of our heuristic is also to sequentially optimize each job's estimated completion time by linear programming. Algorithm 8 shows the details. Most parts of Algorithm 8 are the same as the optimization for *AMF*. We illustrate the different parts from the optimization of *AMF*. The inputs include $\mathcal{J}_{seq}$ and $\vec{A}$. $\mathcal{J}_{seq}$ is a sequence of all the jobs, which specifies the order of optimization. $\vec{A} = \langle A_1, A_2, ..., A_n \rangle$ is the job-wise allocation vector obtained from the *GAMF* allocation $\mathbf{A}_{n \times m \times g}$ returned by Algorithm 7. This will be used as constraints to ensure that the optimized allocation also meets *GAMF*. In each round, the first job J_l in $\mathcal{J}_{seq}$ is set as the optimization target (line 4). As each task has multiple available sites to allocate, and each job's demand on each site is not known. Thus, we consider estimating the job completion time according to task groups. If J_l is allocated an amount $\sum_{j:S_j \in G_k} a_{ljk}$ of resources at all the available sites of task group G_k (where a_{ljk} is the amount of resources allocated to J_l by site S_j for task group G_k), then the completion time of all its tasks in task group G_k is proportional to $\frac{d_{lk}}{\sum_{j:S_j \in G_k} a_{ljk}}$, where d_{lk} is the remaining number of J_l's tasks in task group G_k. Thus, the completion time of job J_l can be estimated as $\max_{G_k \in \mathcal{G}} \frac{d_{lk}}{\sum_{j:S_j \in G_k} a_{ljk}}$. To optimize the estimated completion time by linear programming, we also convert the objective to a linear one. We rewrite each task group's allocation $\sum_{j:S_j \in G_k} a_{ljk}$ as $d_{lk} \cdot y_l + z_{lk}$, where $y_l \geq 0$ is a job-wise variable and $z_{lk} \geq 0$ is an auxiliary variable. Then, when y_l is maximized, J_l's estimated completion time is minimized. Similarly, to optimize a *SIG-GAMF* allocation, we can add the sharing incentive constraint (5.24) for each job $J_i \in \mathcal{J}_{seq}$ to the linear program of Algorithm 8.

5.5 Experimental Setup

We conduct simulations with similar settings to Chapter 4.

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 [3, 20]. 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

Algorithm 8: Optimization for *GAMF*

Input: a sequence of all jobs $\mathcal{J}_{seq}$ and a job-wise allocation vector $\vec{A}$;
Output: an optimized allocation $\mathbf{A}'_{n\times m\times g}$;
while $\mathcal{J}_{seq} \neq \emptyset$ **do**
    $J_l \leftarrow$ the first job in $\mathcal{J}_{seq}$;
    solve the following linear program:
    maximize y_l
    subject to: $\forall S_j \in \mathcal{M} : \sum_{i=1}^{n} \sum_{k:S_j \in G_k} a_{ijk} \leq u_j$;
    $\forall J_i \notin \mathcal{J}_{seq}, S_j \in \mathcal{M}, G_k \in \mathcal{G} : a_{ijk} = a'_{ijk}$;
    $\forall J_i \in \mathcal{J}_{seq}, G_k \in \mathcal{G} : 0 \leq \sum_{j:S_j \in Gk} a_{ijk} \leq d_{ik}$;
    $\forall J_i \in \mathcal{J}_{seq} : \sum_{k=1}^{g} \sum_{j:S_j \in G_k} a_{ijk} = A_i$;
    $\forall G_k \in \mathcal{G} : \sum_{j:S_j \in G_k} a_{ljk} = d_{lk} \cdot y_l + z_{lk}$;
    $y_l \geq 0$;
    $\forall G_k \in \mathcal{G} : z_{lk} \geq 0$;
    for each $S_j \in \mathcal{M}$ and $G_k \in \mathcal{G}$ **do**
        $a'_{ljk} \leftarrow a_{ljk}$;
    end for
    remove J_l from $\mathcal{J}_{seq}$;
end while
return $\mathbf{A}'_{n\times m\times g}$

Pareto distribution with parameter $\beta = 1.259$ [6] and a mean of 2 seconds. The Google trace was collected on a cluster at Google for one month [1, 53]. 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 mean task duration is 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 number of sites is set at 10. The sites are denoted as $S_1, S_2, ..., S_{10}$. 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 task distribution is expected to be 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. In the experiments, we set the number of available sites at 2.

Resource Allocation Policies: We implement various *GAMF* policies. The Basic *GAMF* runs Algorithm 7 to compute the resource allocation. The Optimized *GAMF* runs Algorithm 8 after Algorithm 7 to optimize the job completion times. The Basic *SIG-GMAF* and Optimized *SIG-GAMF* further guarantee the sharing incentive property. For all the policies, we recompute the resource allocation among active jobs whenever a new job arrives or a running job completes all its tasks at a site. For comparison purpose, we also implement the *AMF* policies, *i.e.,*, Basic *AMF*, Optimized *AMF*, Basic *SIG-AMF* and Optimized *SIG-AMF*. In these *AMF* policies, we stipulate that each task of a job must be executed at its associated site.

Performance Metrics: We evaluate the fairness of resource allocation for jobs with tasks executable at multiple sites and the effectiveness of the optimization in terms of job response times. Thus, we study the standard deviation of jobs' aggregate resource allocations and the average job response time. The methods of computing the standard

deviation and average job response time are the same as those in the experiments of Chapter 4. The lower the standard deviation, the more balanced the resource allocation among jobs. Since the number of computing slots allocated to a job at any time is integer valued and a computing slot can be allocated to a new task only after its current task is finished, the standard deviation of the instantaneous aggregate allocations of active jobs is not necessarily equal to that of the allocation matrix output by the programming algorithms.

5.6 Experimental Results

The general trends of the experimental results are similar to those in Chapter 4. Figures 5.2 and 5.1 show the cumulative distribution of the standard deviation for the aggregate resources received by the active jobs for the Google and Facebook traces when the Zipf parameter of task distribution varies from 0 to 2 and the system utilization is 60% (the trends for other system utilizations are similar). A point (x, y) on the curve means that the standard deviation is no more than x for y portion of the time. These results show that the job-wise aggregate allocations are more balanced in *GAMF* than in *AMF*. The nature of *AMF* and *GAMF* is to ensure the job-wise resource allocations are as even as possible. With more available sites, each job has more chances to adjust the allocation among sites to get a more balance result.

Figures 5.4 and 5.3 show the average job response time for the Google and Facebook traces with different system utilizations and skewness of task group distribution. As we can see from the figures, all the variants of *GAMF* perform better than the corresponding variants of *AMF* in terms of average job response time. This is because when it comes to *GAMF*, multiple sites are available for executing each task which could enable more balanced allocation and shorten its completion time, thereby reducing the job response time.

Basic *GAMF* only ensures that the job-wise allocation vector is max-min fair. In an arbitrary allocation output by Basic *GAMF* (Algorithm 7), the amount of resources received by a job for different task groups can be quite unbalanced. As a result, some task groups can be finished much faster than other task groups in the same job, leading

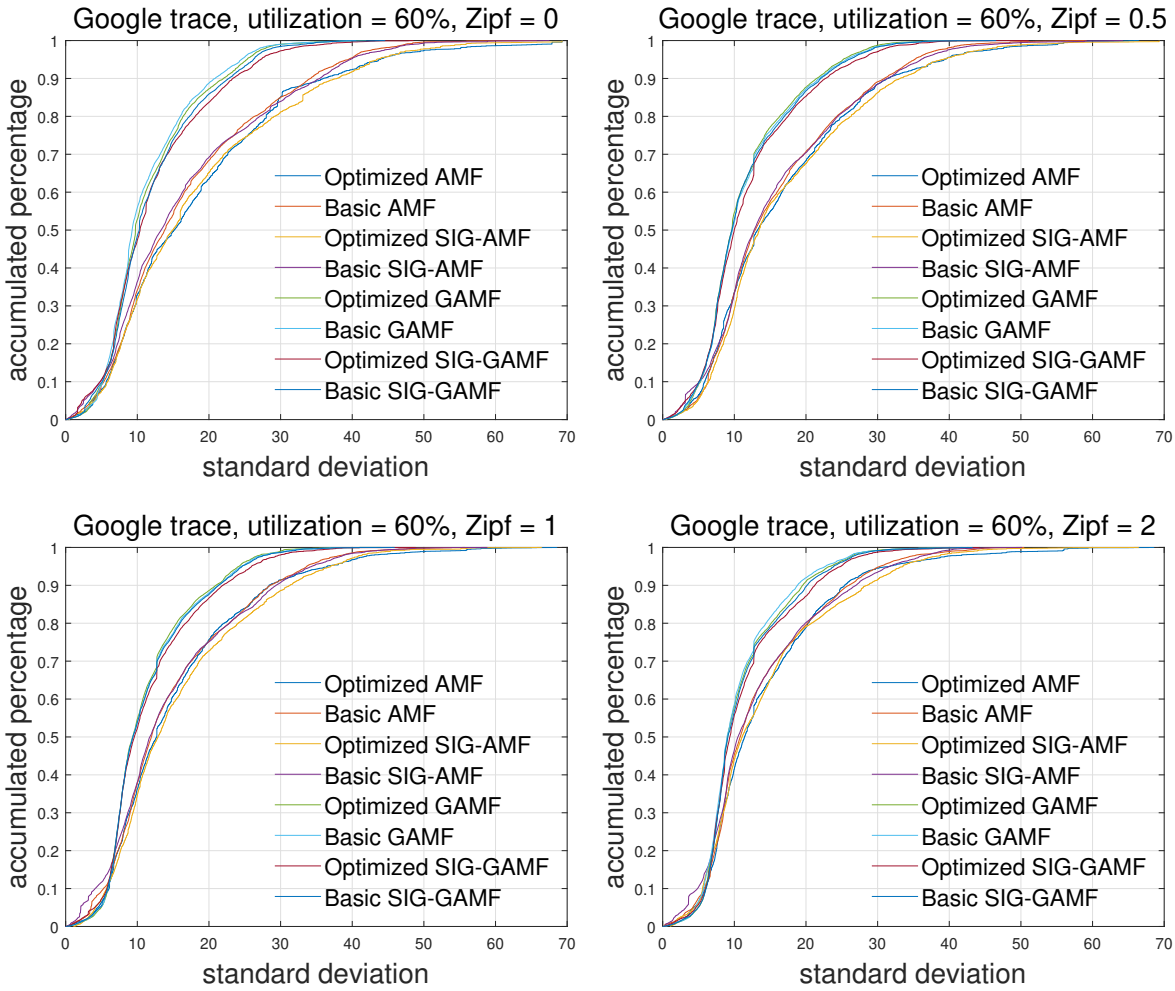


Figure 5.1: Cumulative distribution of standard deviation for Google trace (2 available sites, utilization = 60%). The curves are clustered in two groups. All the *AMF* policies are in the group closer to the point (0, 1), and all the *GAMF* policies are in the group further from the point (0, 1).

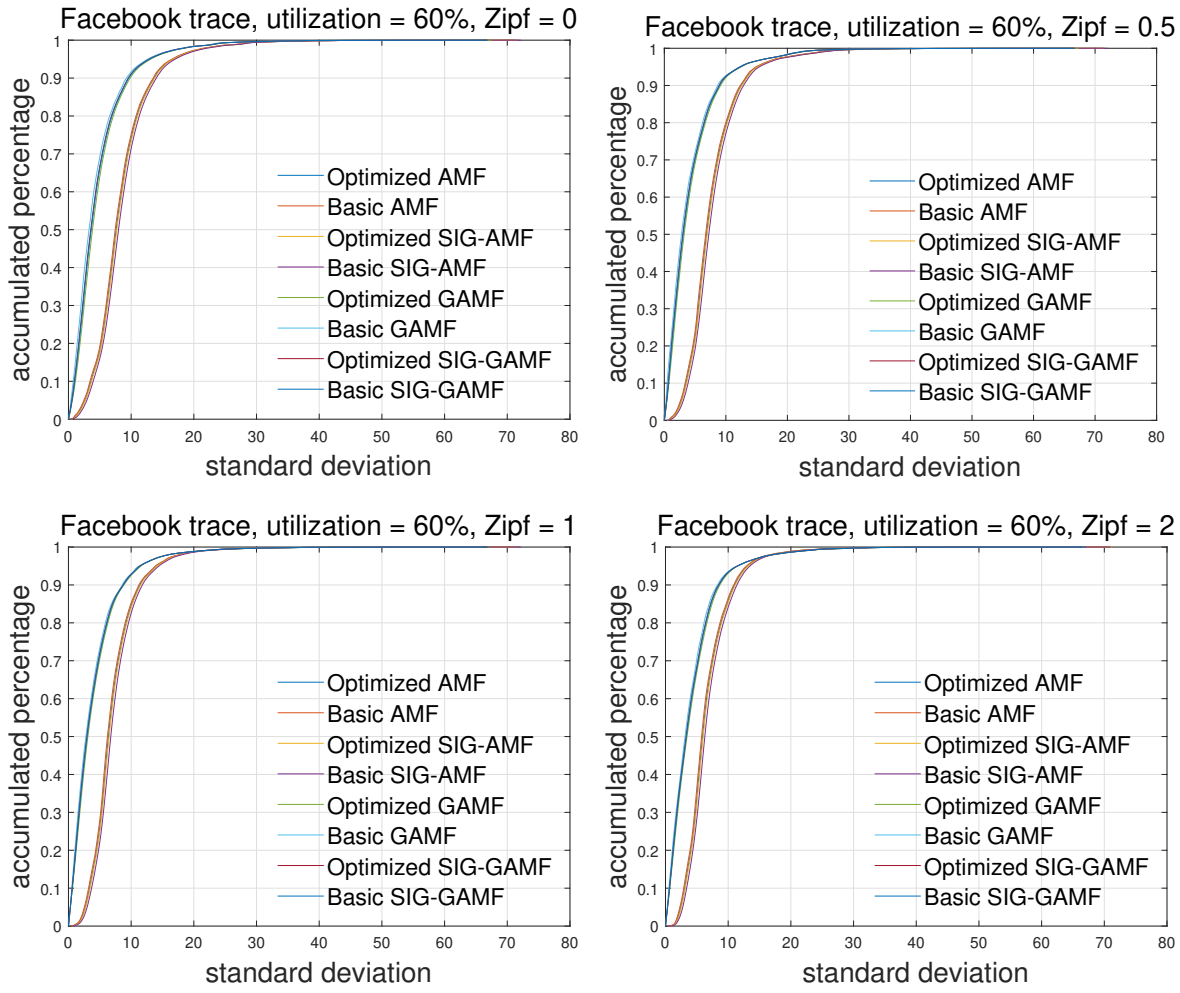


Figure 5.2: Cumulative distribution of standard deviation for Facebook trace (2 available sites, utilization = 60%). The curves are clustered in two groups. All the *AMF* policies are in the group closer to the point (0, 1), and all the *GAMF* policies are in the group further from the point (0, 1).

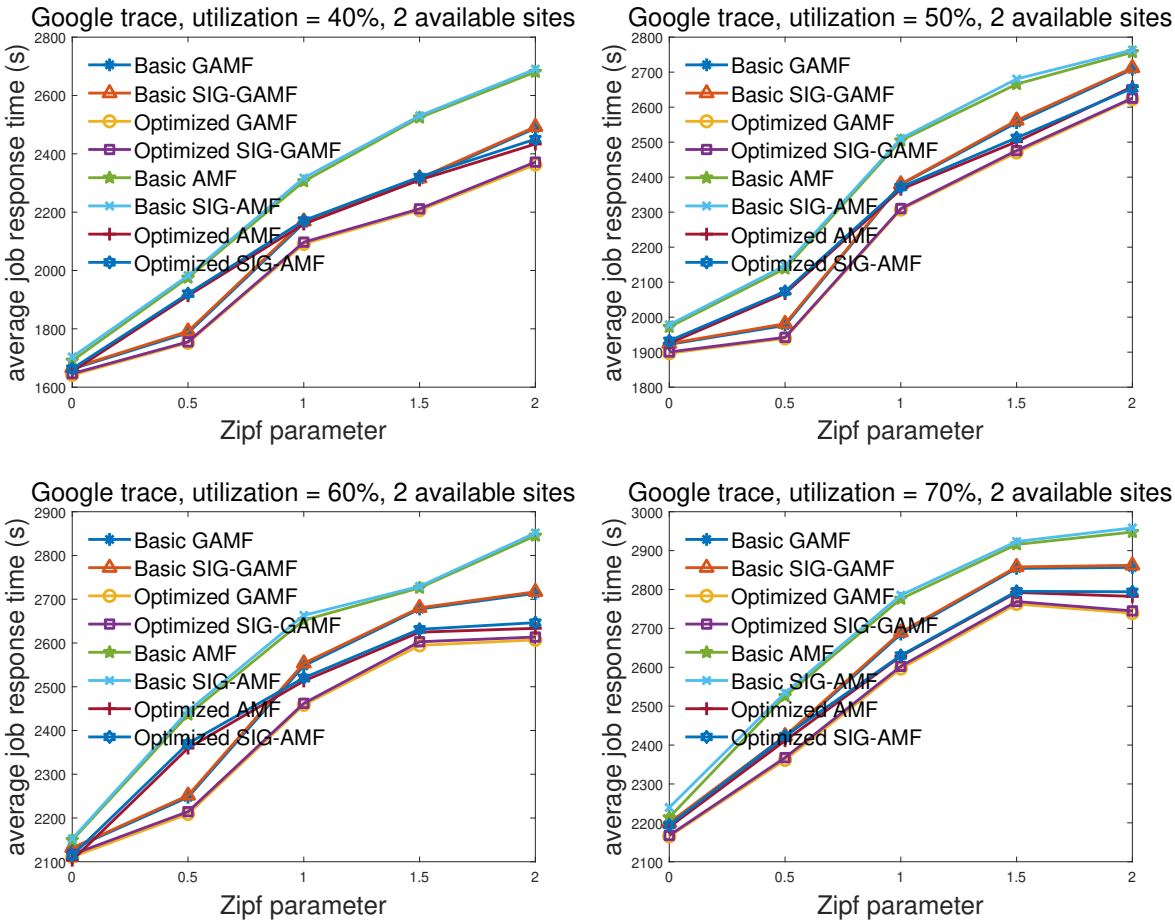


Figure 5.3: Average job response time for Google trace (2 available sites)

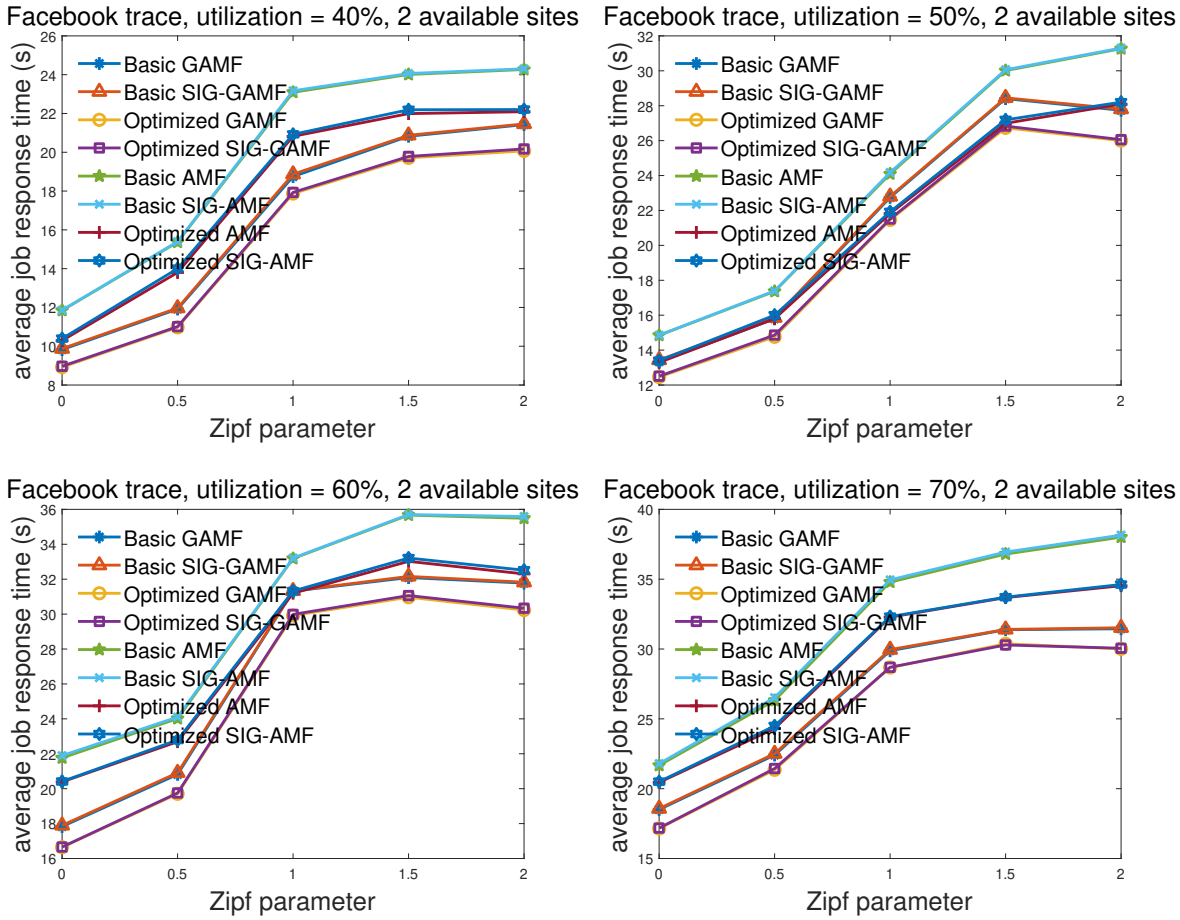


Figure 5.4: Average job response time for Facebook trace (2 available sites)

to longer job response time. By refining the task group level allocation with Algorithm 8, Optimized *GAMF* allocates more resources to larger task groups and further improves average job response time over Basic *GAMF* as shown in Figures 5.4 and 5.3.

SIG-GAMF produces almost the same standard deviation of job resource allocation and the same average job response time as *GAMF* for the traces tested. This indicates that most of the resource allocations generated by *GAMF* satisfy the sharing incentive property. Hence, enforcing the sharing incentive property in *GAMF* does not change the resource allocations substantially.

5.7 Summary

In this chapter, we have extended *AMF* to *GAMF* for a more general scenario in which jobs have tasks that can be executed at multiple sites. The assignment of each task to one of its available sites gives an additional dimension of freedom in resource allocation. As a result, the design and analysis of the *GAMF* resource allocation policy becomes more complicated than that of *AMF*. We prove that *GAMF* satisfies the properties of Pareto Efficiency, envy-freeness and strategy-proofness for fair resource allocation. The additional dimension of freedom makes it nearly impossible to prove the strategy-proofness for *GAMF* in the same way as that for *AMF*. We develop a new method to prove that *GAMF* is strategy-proof. Though *GAMF* does not satisfy the sharing incentive property, we enhance *GAMF* to guarantee the sharing incentive property. We present algorithms to implement *GAMF* and to optimize the job response times under *GAMF*. Experimental results show that by taking advantage of the additional freedom in assigning each task to one of its available sites, *GAMF* can achieve more balanced resource allocation and reduce average job response time compared to *AMF*.

Chapter 6

Conclusion and Future Work

In this thesis, we have studied three resource allocation problems for distributed job execution: task assignment and scheduling for improving efficiency of distributed job execution, max-min fair resource allocation for distributed job execution, and generalized max-min fair resource allocation for jobs with tasks executable at multiple sites. The main contributions of the thesis include the heuristics for improving the efficiency of distributed job execution and the definition of fairness policies for resource allocation. The fairness policies ensure that users will not envy each other when competing for resources and they cannot be allocated more resources by lying about their resource demands.

In Chapter 3, we have studied task assignment and scheduling for improving efficiency of 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 several integrated solutions to carry out task assignment and job scheduling together. Experimental results show that the integrated 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.

In Chapter 4, we have studied several resource allocation policies for distributed job execution. We prove that *AMF* satisfies widely recognized properties of Pareto efficiency, envy-freeness and strategy-proofness for fair resource allocation, but the vanilla version of *AMF* does not satisfy the sharing incentive property. We propose an enhanced version of *AMF* to guarantee the sharing incentive property. We present algorithms to implement

AMF as well as to optimize the job response times under *AMF*. Experimental results show that *AMF* can considerably reduce unbalanced resource allocation and improve average job response time compared to *IMF* which conducts max-min fair allocation at each site separately, and the improvements are more significant when the workload distribution of jobs among sites is more skewed.

In Chapter 5, we have extended *AMF* to *GAMF* to handle jobs with tasks executable at multiple sites. We prove that *GAMF* satisfies the properties of Pareto Efficiency, envy-freeness and strategy-proofness for fair resource allocation. Though *GAMF* does not satisfy the sharing incentive property, we propose an enhanced version to guarantee the sharing incentive property. We present algorithms to implement *GAMF* and to optimize the job response times under *GAMF*. Experimental results show that *GAMF* can reduce average job response time and achieve more balanced resource allocation compared to *AMF*.

Following the work described in this thesis, there are several directions to continue the research on resource allocation fairness and efficiency for distributed job execution. One potential limitation of the fairness policies developed in this thesis is that they abstract resources in the form of computing slots, which simplifies the consideration to essentially one resource type. With the development of distributed applications, there can be more diverse types of resource demands from users. It will be useful to extend the fairness policies for distributed job execution to multiple types of resources. Another potential limitation is that the programming algorithms for computing fair resource allocations involve solving linear programs, which may be time-consuming. To improve the implementation efficiency, we may model fair resource allocations as graph-based problems such as parametric flow problems and use parametric flow algorithms to derive fair resource allocations. Moreover, we can work on other variants of fair resource allocations. For example, we can study the fairness among the accumulated resources received by individual jobs over any period of time. In addition, resource allocation yields a tradeoff between fairness and efficiency. To optimize the job response times, we prefer concentrating the resource allocation on a few jobs to complete them soonest possible. On the other hand, to ensure fairness, we prefer balancing the resource allocations among all the jobs. We can study the fairness-efficiency tradeoff and investigate the best operating point along this tradeoff for distributed job execution.

Finally, learning-based approaches are interesting to investigate in the context of distributed job execution. There are recent works applying reinforcement learning to job allocation and scheduling [47, 65], but most of them are for relatively simple scenarios. For example, each job is to be executed on a single server, each server runs a single job at a time, and jobs are scheduled following simple rules like FIFO, where the system state and job demands are easy to describe and the number of possible allocation actions for a particular job is quite limited. In our context of distributed job execution, the state of the system needs to record all the outstanding jobs, each including its task groups, the available sites of each task group, and the number of outstanding tasks in each task group. The number of possible task groups (possible combinations of available sites) is exponential with respect to the number of sites. 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 site. The number of possible orders to execute jobs is factorial with respect to the number of jobs. Thus, the action space is also exponentially large. With huge problem sizes for the context of distributed job execution, it is difficult to apply off-the-shelf reinforcement learning algorithms to our scheduling problem. Additional techniques in state and action representations as well as training techniques will need to be developed to design efficient reinforcement learning algorithms workable for our problem.

During my research journey, I experienced elation and frustration. There were delays along the way due to unforeseen circumstances. I have realized that conducting research is not a quick process and good research work needs a lot of skills, patience and intelligence. I have learned to be more cautious and studious in the research journey.

References

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

[2] "Shortest remaining time," https://en.wikipedia.org/wiki/Shortest_remaining_time.

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

[4] A. K. Agrawala and R. M. Bryant, "Models of memory scheduling," *ACM SIGOPS Operating Systems Review*, vol. 9, no. 5, pp. 217–222, 1975.

[5] F. Ahmad, S. A. Mahmud, G. M. Khan, and F. Z. Yousaf, "Shortest remaining processing time based schedulers for reduction of traffic congestion," in *International Conference on Connected Vehicles and Expo (ICCVE)*. IEEE, 2013, pp. 271–276.

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

[7] S. Banerjee, A. Budhiraja, and A. L. Puha, "Heavy traffic scaling limits for shortest remaining processing time queues with heavy tailed processing time distributions," *arXiv*, p. arXiv:2003.03655.

[8] N. Bansal and M. Harchol-Balter, "Analysis of srpt scheduling: Investigating unfairness," in *Proceedings of ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2001, pp. 279–290.

[9] J. C. Bennett and H. Zhang, "Wf/sup 2/q: worst-case fair weighted fair queueing," in *INFOCOM*, vol. 1. IEEE, 1996, pp. 120–128.

[10] A. Benoit, L. Marchal, J.-F. Pineau, Y. Robert, and F. Vivien, "Offline and online master-worker scheduling of concurrent bags-of-tasks on heterogeneous platforms," in *International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

[11] J. Berger, B. P. Cohen, T. L. Conner, and M. Zelditch Jr, "Status characteristics and expectation states: a process model," Standford Sociology, 2015, Tech. Rep.

[12] D. P. Bertsekas, R. G. Gallager, and P. Humblet, *Data networks*. Prentice-Hall International New Jersey, 1992, vol. 2.

[13] T. Bonald and J. Roberts, "Multi-resource fairness: Objectives, algorithms and performance," *arXiv*, pp. arXiv–1410, 2014.

[14] ——, "Multi-resource fairness: Objectives, algorithms and performance," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1. ACM, 2015, pp. 31–42.

[15] Z. Cao and E. W. Zegura, "Utility max-min: An application-oriented bandwidth allocation scheme," in *INFOCOM)*, vol. 2. IEEE, 1999, pp. 793–801.

[16] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng, "Group ratio round-robin: O (1) proportional share scheduling for uniprocessor and multiprocessor systems," in *USENIX Annual Technical Conference*, 2005, pp. 337–352.

[17] R.-S. Chang, C.-Y. Lin, and C.-F. Lin, "An adaptive scoring job scheduling algorithm for grid computing," *Information Sciences*, vol. 207, pp. 79–89, 2012.

[18] 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*, pp. 1–1, 2018.

[19] ——, "Scheduling jobs across geo-distributed datacenters with max-min fairness," *IEEE Transactions on Network Science and Engineering*, vol. 6, no. 3, p. 488, 2019.

[20] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating MapReduce performance using workload suites," in *IEEE MASCOTS 2011*. IEEE, 2011.

[21] A. Coluccia, A. D'Alconzo, and F. Ricciato, "On the optimality of max–min fairness in resource allocation," *annals of telecommunications-annales des télécommunications*, vol. 67, no. 1-2, pp. 15–26, 2012.

[22] R. I. Davis, A. Burns, and W. Walker, "Guaranteeing timing constraints under shortest remaining processing time scheduling," in *Proceedings Ninth Euromicro Workshop on Real Time Systems*. IEEE, 1997, pp. 88–93.

[23] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 1–12, 1989.

[24] D. Dolev, D. G. Feitelson, J. Y. Halpern, R. Kupferman, and N. Linial, "No justified complaints: On fair sharing of multiple resources," in *proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM, 2012, pp. 68–75.

[25] ——, "No justified complaints: On fair sharing of multiple resources," *arXiv*, p. arXiv:1106.2673.

[26] D. G. Down, H. C. Gromoll, and A. L. Puha, "Fluid limits for shortest remaining processing time queues," *Mathematics of Operations Research*, vol. 34, no. 4, pp. 880–911, 2009.

[27] L. Ėchrage, "A proof of the optimality of the shortest processing remaining time discipline," *Operations Research*, vol. 16, pp. 678–690, 1968.

[28] S. Ghanbari and M. Othman, "A priority based job scheduling algorithm in cloud computing," *Procedia Engineering*, vol. 50, no. 0, pp. 778–785, 2012.

[29] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *Nsdi*, vol. 11, no. 2011, 2011, pp. 24–24.

[30] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Choosy: Max-min fair sharing for datacenter jobs with constraints," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 365–378.

[31] H. C. Gromoll and M. Keutel, "Invariance of fluid limits for the shortest remaining processing time and shortest job first policies," *Queueing Systems*, vol. 70, no. 2, pp. 145–164, 2012.

[32] H. C. Gromoll, Ł. Kruk, and A. L. Puha, "Diffusion limits for shortest remaining processing time queues," *Stochastic Systems*, vol. 1, no. 1, pp. 1–16, 2011.

[33] 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," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1259–1270, 2014.

[34] E. L. Hahne, "Round-robin scheduling for max-min fairness in data networks," *IEEE Journal on Selected Areas in communications*, vol. 9, no. 7, pp. 1024–1039, 1991.

[35] M. Hajjat, D. Maltz, S. Rao, and K. Sripanidkulchai, "Dealer: application-aware request splitting for interactive cloud applications," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012, pp. 157–168.

[36] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal, "Size-based scheduling to improve web performance," *Transactions on Computer Systems (TOCS)*, vol. 21, no. 2, pp. 207–233, 2003.

[37] Y. T. Hou, H.-Y. Tzeng, and S. S. Panwar, "A generalized max-min rate allocation policy and its distributed implementation using the abr flow control mechanism," in *INFOCOM*, vol. 3. IEEE, 1998, pp. 1366–1375.

[38] Z. Hu, K. Wu, and J. Huang, "An utility-based job scheduling algorithm for current computing cloud considering reliability factor," in *International Conference on Computer Science and Automation Engineering.* IEEE, 2012, pp. 296–299.

[39] X. L. Huang and B. Bensaou, "On max-min fairness and scheduling in wireless ad-hoc networks: Analytical framework and implementation," in *MOBIHOC*, 2001, pp. 221–231.

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

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

[42] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of IGOPS 22nd symposium on Operating systems principles*, 2009, pp. 261–276.

[43] J. Y.-T. Leung, H. Li, and M. Pinedo, "Scheduling orders for multiple product types to minimize total weighted completion time," *Discrete Applied Mathematics*, vol. 155, no. 8, pp. 945–970, 2007.

[44] J. Li, L. Feng, and S. Fang, "An greedy-based job scheduling algorithm in cloud computing." *J. Softw.*, vol. 9, no. 4, pp. 921–925, 2014.

[45] L. Li, "An optimistic differentiated service job scheduling system for cloud computing service users and providers," in *International Conference on Multimedia and Ubiquitous Engineering*. IEEE, 2009, pp. 295–299.

[46] M. Lin, A. Wierman, and B. Zwart, "Heavy-traffic analysis of mean response time under shortest remaining processing time," *Performance Evaluation*, vol. 68, no. 10, pp. 955–966, 2011.

[47] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM workshop on hot topics in networks*. ACM, pp. 50–56.

[48] P. Marbach, "Priority service and max-min fairness," in *IEEE Computer and Communications Societies*, vol. 1. IEEE, 2002, pp. 266–275.

[49] 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," *Operations Research Letters*, vol. 38, no. 5, pp. 390–395, 2010.

[50] R. Mohanty, H. Behera, K. Patwari, and M. Dash, "Design and performance evaluation of a new proposed shortest remaining burst round robin (srbrr) scheduling algorithm," in *Proceedings of International Symposium on Computer Engineering & Technology*, vol. 17, 2010, pp. 126–137.

[51] 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. ACM, 2015, pp. 421–434.

[52] B. Radunović and J.-Y. L. Boudec, "A unified framework for max-min and min-max fairness with applications," *IEEE/ACM Transactions on Networking (TON)*, vol. 15, no. 5, pp. 1073–1083, 2007.

[53] 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 *ACM Symposium on Cloud Computing (SoCC)*. ACM, 2012.

[54] J. Ros and W. K. Tsai, "A theory of convergence order of maxmin rate allocation and an optimal protocol," in *INFOCOM*, vol. 2. IEEE, 2001, pp. 717–726.

[55] D. Rubenstein, J. Kurose, and D. Towsley, "The impact of multicast layering on network fairness," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 27–38, 1999.

[56] S. Sarkar and L. Tassiulas, "Fair bandwidth allocation for multicasting in networks with discrete feasible set," *IEEE Transactions on Computers*, vol. 53, no. 7, pp. 785–797, 2004.

[57] D. R. Smith, "A new proof of the optimality of the shortest remaining processing time discipline," *Operations Research*, vol. 26, no. 1, pp. 197–199, 1978.

[58] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *17th IEEE Real-Time Systems Symposium*. IEEE, 1996, pp. 288–299.

[59] L. Tassiulas and S. Sarkar, "Maxmin fair scheduling in wireless networks," in *IEEE Computer and Communications Societies*, vol. 2. IEEE, 2002, pp. 763–772.

[60] P. Varalakshmi, A. Ramaswamy, A. Balasubramanian, and P. Vijaykumar, "An optimal workflow based scheduling and resource allocation in cloud," in *International Conference on Advances in Computing and Communications*. Springer, 2011, pp. 411–420.

[61] 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 *NSDI*, vol. 7, no. 7.2, 2015, pp. 7–8.

[62] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Conference on Operating Systems Design and Implementation*. USENIX, 1994, pp. 1–11.

[63] W. Wang, B. Li, and B. Liang, "Dominant resource fairness in cloud computing systems with heterogeneous servers," *arXiv*, p. arXiv:1308.0083, 2013.

[64] ——, "Dominant resource fairness in cloud computing systems with heterogeneous servers," in *INFOCOM*. IEEE, 2014, pp. 583–591.

[65] D. Yi, X. Zhou, Y. Wen, and R. Tan, "Efficient compute-intensive job allocation in data centers via deep reinforcement learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1474–1485, 2020.

[66] 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 *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.