On Max-Min Fairness of Completion Times for Multi-Task Job Scheduling

Mehrnoosh Shafiee, Javad Ghaderi
Electrical Engineering Department, Columbia University

***Abstract*—We study the max-min fairness of multi-task jobs in distributed computing platforms. We consider a setting where each job consists of a set of parallel tasks that need to be processed on different servers, and the job is completed once all its tasks finish processing. Each job is associated with a utility which is a decreasing function of its completion time, and captures how sensitive it is to latency. The objective is to schedule tasks in a way that achieves max-min fairness for jobs' utilities, i.e., an optimal schedule in which any attempt to improve the utility of a job necessarily results in hurting the utility of some other job with smaller or equal utility.**

We first show a strong result regarding NP-hardness of finding the max-min fair vector of job utilities. The implication of this result is that achieving max-min fairness in many other distributed scheduling problems (e.g., coflow scheduling) is NP-hard. We then proceed to define two notions of approximation solutions: one based on finding a certain number of elements of the max-min fair vector, and the other based on a single-objective optimization whose solution gives the max-min fair vector. We develop scheduling algorithms that provide guarantees under these approximation notions, using dynamic programming and random perturbation of tasks' processing times. We verify the performance of our algorithms through extensive simulations, using a real traffic trace from a large Google cluster.

***Index Terms*—Scheduling Algorithms, Max-Min Fairness, Lexicographic Optimization, Data Centers**

I. Introduction

Distributed computing platforms, such as MapReduce [1], Dryad [2], Spark [3], etc., have been widely adapted for large-scale data processing in cloud and computing clusters. The data set is typically distributed among a set of servers, and processed by executing a job consisting of a set of tasks in servers. The tasks are typically processed in the servers where their input data is stored (a.k.a data locality) [1]. The collective behavior of tasks is more important than each of the tasks individually, as the job can be completed, or moved to another computation stage, *only when all of its tasks finish their processing* [1]–[3].

Jobs from a wide range applications and different users can coexist in the same cluster, and often have diverse tasks and processing requirements. *Efficient* and *fair* allocation of the cluster's resources among jobs is crucial to guarantee their timely completions. This has been amplified by the increasing complexity of workloads, i.e., from traditional batch jobs, to queries, graph processing, streaming, machine learning jobs, etc., that all need to share the same cluster, and often have very different latency and priority requirements. For example, analysis of a Google cluster's trace in [4] shows a diverse mix of jobs in the same cluster, ranging from latency-tolerant jobs ($\sim 24\%$) to latency-sensitive jobs ($\sim 42\%$).

Fair allocation of resources in shared clusters among applications and organizations has been studied in, e.g., [5]–[7], where the cluster's resources are usually divided among different applications through some notion of fairness, e.g. DRF (Dominant Resource Fair) [5]. The scheduler then manages queues of tasks for applications and schedules their tasks. For example, Hadoop [8] reserves resources by launching *containers* or *virtual machines* in servers. Each container reserves memory and CPU for *processing a task at a time*. The Hadoop scheduler uses FIFO scheduling or memory-based DRF [9]. However, these schedulers ignore the completion times of jobs and their latency requirements when allocating resources. Assigning priorities can alleviate this problem, however priorities are typically assigned to applications manually [10], and it is not clear how to assign priorities to *jobs* (and their tasks) dynamically, based on the existing jobs in the cluster and their sensitivities to latency. Further, application priority in Hadoop is supported only for FIFO scheduling [10].

Despite the vast research on scheduling algorithms (*see Related Work*), theoretical study of fairness with focus on sensitivities of jobs to latency is very limited. Moreover, prior work is mainly based on simple models that assume each job is only one task (ignoring dependency among tasks and their collective impact on the job's completion time), or tasks are processed on any server arbitrarily (ignoring data locality).

In this paper, we consider a multi-task job scheduling model that captures such features. Each job consists of a set of tasks whose completion time is determined by the completion time of its last task. As in [11], [12], to capture latency-sensitivity, we consider a utility for each job as a function of its completion time. For example, a highly latency-sensitive job can specify a utility function that decays rapidly to zero as its completion time increases. We consider the notion of *max-min fairness* which is one of the most widely used resource allocation mechanisms [13]–[15]. Our objective is to schedule tasks in a way that achieves *max-min* fairness among jobs' utilities, i.e., maximize the worst utility across all the jobs, then maximize the second-worst utility without affecting the worst utility, and so on. We refer to this problem as *max-min fair scheduling*. Note that this implies that at the optimal solution, we *cannot* increase the utility of any job without hurting the utility of some job with smaller or equal utility.

ISBN 978-3-903176-28-7
This research was supported by NSF grants CNS-1717867 and CNS-1652115.

We also would like to mention that the max-min fair scheduling problem for our multi-task job model is of interest from theoretical point of view. As we see later, our model can be reduced to the three scheduling problems considered in the literature to achieve max-min fairness for jobs and coflows[1] considered in [12], [17], [18]. Hence, all the three problems are at least as hard as our problem.

A. Related Work

There has been much work on fair scheduling in data centers, e.g. [5]–[7], [11], [12], [17], [19]–[21]. They mostly consider fair resource allocation to guarantee properties such as sharing-incentive among users of a shared cloud [5]–[7], with little focus on the sensitivity of jobs to their completion times, or consider heuristics for different notions of fairness for maximizing total utility [19], meeting deadlines [21], or fair resource assignment to each job [20]. Generating proper utility functions based on jobs' priorities and completion times was studied in [11]. In [19], a Risk-Reward heuristic was presented where scheduling decisions are made based on the cost of reallocating resources and future utility gain. The max-min fairness of job utilities were studied in [12], [17], however, their models assume each job has only one task and the cloud cluster is one large pool for each resource type. Moreover, in their solution, a job can be allocated different resource types in different unrelated time slots, as opposed to having all its required resources available at the same time. Further, despite their plausible algorithms that try to solve the problem optimally, we show in this paper that the problems are NP-hard in a strong sense.

Our model is closely related to the *concurrent open shop* model [22]–[25] in scheduling literature. Minimizing the (weighted) average completion time of jobs in this model has been widely studied, with several approximation algorithms in [23]–[25]. However, to the best of our knowledge, there is no theoretical result on max-min fairness in this model.

Max-min fair is one of the most widely used notions of fairness [13]–[15]. Moreover, the use of utilities and the network utility maximization for rate allocation in communication networks has been extensively studied (e.g. see [26] and references therein). However, the results cannot be extended to max-min fair job scheduling in data centers. The max-min fair optimization is not a single-objective optimization, as we aim to optimize a vector of objective functions (utilities) in the sense of max-min fairness. Multi-objective optimization programs have been widely studied and different methods have been developed to solve these problems efficiently in special cases [27]–[29]. However, as we show, solving the multi-objective optimization in our setting is a hard problem (NP-hard). In [29], existence and computation of a set of equivalent weights was studied which enable the conversion of a given multi-objective optimization to a single-objective optimization. We use this method in our paper to study the performance of one of our proposed algorithms.

[1]a collection of flows whose completion time is the completion time of the last flow in the collection [16].

We would like to mention that, there exist well-established techniques to estimate tasks' durations to the scheduler, based on the history of prior runs for recurring jobs, using tasks' peak demands, or measuring statistics from the first few tasks in each job, see [30], [31]. Hence, throughout the paper, we assume that tasks' processing times are known.

B. Main Contributions

Our main contributions can be summarized as follows:

- **NP-Hardness of Max-Min Fair Scheduling:** We first show that it is NP-hard to optimally solve the max-min fair scheduling problem. We actually prove a stronger complexity result. Given n multi-task jobs in a cluster of machines, it is NP-hard to find a schedule in which even the first $O(n^\epsilon) \ll n$ number of jobs, for any $\epsilon > 0$, conform to their optimal max-min fair solution. Further, we conclude that the scheduling problems considered in [12], [17], [18] are NP-hard.
- **Approximation Algorithms:** We define two notions of approximation solutions for this problem: one based on finding a constant number of elements of the max-min fair vector, and the other based on a single-objective optimization whose solution gives the max-min fair vector. We develop scheduling algorithms, using dynamic programming and random perturbation of tasks' processing times, that provide guarantees under both notions of approximation solutions.
- **Empirical Evaluation:** We use a real traffic trace from a large Google cluster to verify that our algorithms in fact perform very well compared to other scheduling policies.

II. Model and problem statement

Cluster and Job Model: We consider a cluster of m machines, denoted by the set $M = \{1, ..., m\}$. Each machine can be thought of as a container or virtual machine [8] that can process one task at a time. There is a collection of n jobs, denoted by the set $N = \{1, ..., n\}$. Each job $j \in N$ consists of up to m different tasks that need to be processed on different machines. Each task requires a specific processing time from its corresponding machine[2]. Specifically, the task of job j on machine i, denoted by task (i, j), requires a processing time $p_{ij} \geq 0$ from machine i. For each job j, we use $M_j \subseteq M$ to denote the subset of machines that contain tasks of job j, i.e., $p_{ij} > 0$ for each $i \in M_j$. Without loss of generality, we assume processing times of all non-zero tasks are integer numbers and the smallest processing time is at least one. This can be done by defining a proper time unit and representing the tasks' processing times using integer multiples of this unit.

Tasks are independent of each other in the sense that tasks of the same job can be processed in parallel on their corresponding machines. However, a job is completed only when all of its tasks finish their processing. Define C_{ij} to be

[2]In the case that a job has multiple tasks on a specific machine, we can view them as a single task with processing time equal to the cumulative original tasks' processing times.

the completion time of task (i, j). Then, the completion time of job j, denoted by C_j, is given by

$$C_j = \max_{i \in M_j} C_{ij}. \tag{1}$$

This model is known as the *concurrent open shop* problem [22]–[25] in scheduling literature. The total time that it takes to complete all the jobs in N is called *makespan* and is denoted by $\tau^{(N)}$. Note that by definition $\tau^{(N)} = \max_{j \in N} C_j$. It is easy to see that any valid schedule that does not leave a machine idle, unless it has completed all its corresponding tasks, achieves the optimal makespan which is equal to

$$\tau^{(N)} = \max_{i \in M} \sum_{j \in N} p_{ij}. \tag{2}$$

Max-Min Fair Objective: As in [11], [12], we assume each job j specifies a utility $U_j(C_j)$, which is a function $U_j(\cdot)$ of its completion time C_j, and captures its sensitivity to its completion time. Since each job prefers an earlier completion time, the utility function is assumed to be decreasing (with respect to the completion time). We further assume that the utility function is Lipschitz continuous (i.e., its first derivative is bounded). To define our max-min fairness, we first define the lexicographic order for two given vectors [32], as follows.

Definition 1 (Lexicographic Order)**.** *Let $X = (X_1, \cdots, X_k)$ and $Y = (Y_1, \cdots, Y_k)$ be two vectors of length k. Sort elements of X and Y in a non-decreasing order and denote the corresponding vectors by $\bar{X}$ and $\bar{Y}$, respectively. We write $X \succ Y$, and say X is lexicographically greater than Y, if $\bar{X}_i > \bar{Y}_i$ for the first i that $\bar{X}_i$ and $\bar{Y}_i$ differ. Consequently, we write $X \succeq Y$ and say vector X is not lexicographically smaller than Y if $\bar{X} = \bar{Y}$ or $X \succ Y$.*

Our objective is to schedule jobs (their tasks) in a way that achieves the max-min fairness across the vector of jobs' utilities. In other words, we wish to maximize the worst (minimum) job utility across all the jobs, and then sequentially maximize the next-worst utility without affecting the previous-worst utility, and so on. Formally, let $C = (C_1, \cdots, C_N)$ be the vector of completion times of jobs in set N and define $U(C) = (U_1(C_1), \cdots, U_N(C_N))$. We seek to schedule jobs in a way that the optimal completion time vector, denoted by $C^\star$, has the property that the vector $U(C^\star)$ is lexicographically greater than $U(C)$ for any other valid scheduling of jobs with completion time vector C, i.e., $U(C^\star) \succeq U(C)$. Note that by Definition 1, the sorted optimal vector $\bar{U}(C^\star)$ is unique.

Preemptive vs Non-preemptive Scheduling: The scheduling algorithm could be preemptive or non-preemptive. In a non-preemptive algorithm, a task cannot be preempted once it starts its processing on its corresponding machine, while in a preemptive algorithm, a task may be preempted and resumed later on the same machine.

III. Lexicographic Max-Min Fair Schedule and NP-hardness

In this section, we first characterize the structure of optimal schedules for max-min fair problem. Then we show a strong result regarding NP-hardness of finding the optimal schedule.

A. Structure of Optimal Schedule

In a non-preemptive schedule (Section II), tasks on each machine are processed according to some order. We say a task is at position l, $l = 1, \cdots, n$, on machine i if it is the l-th task that is completed in machine i. Hence, to fully describe a non-preemptive schedule, it is sufficient to specify a permutation π_i for each machine i, $i \in M$, as formally defined below.

Definition 2 (Permutation of Tasks on Machine i)**.** *Given a set of jobs $N = \{1, 2, \ldots, n\}$ and a valid non-preemptive schedule on machine i, a permutation $\pi_i : N \to \{1, 2, \ldots, n\}$ is a one-to-one mapping of jobs to positions $\{1, 2, \ldots, n\}$ according to which their tasks on machine i are completed.*

Hence $\pi_i(j)$ determines the position of job j's task on machine i. Note that some jobs might not have any tasks on machine i. For these jobs, we consider tasks with zero processing time on machine i. These zero-processing tasks do not contribute to the completion times of jobs and their utilities; nevertheless, including them in Definition 2 will make the future arguments easier. The following theorem characterizes the structure of optimal solution.

Theorem 1. *Any optimal schedule for max-min fair problem can be converted to another optimal schedule in which all the tasks are scheduled in a non-preemptive fashion, according to the same permutation on all the machines.*

Proof Overview. Given any optimal schedule, we construct a non-preemptive schedule, with identical permutation for all machines: Starting from the last job (the job with the largest completion time) in the given solution, we move all its tasks to the end of the schedule in their corresponding machines, and sequentially do this for all the jobs. We omit the details. □

Hence, by Theorem 1, in order to find an optimal solution, it is sufficient to only consider non-preemptive schedules with the same permutation of jobs $\pi_i = \pi$ on all machines $i \in M$.

B. NP-Hardness

Next, we show that finding an optimal solution to the max-min fair scheduling is NP-hard. In fact, we prove a stronger complexity result. Before stating the result, we make the following definition.

Definition 3 ($f(n)$-max-min fair)**.** *Let $\bar{U}(C)$ denote the sorted utility vector corresponding to completion time vector C. We say a solution C is $f(n)$-max-min fair, if the first $f(n)$ elements of $\bar{U}(C)$ match the first $f(n)$ elements of $\bar{U}(C^\star)$ where $C^\star$ is completion time vector for some optimal solution.*

Consider any increasing function $f(n) \leq n$ for which $f^{-1}(n)$ is bounded by a polynomial in n. We show that it is NP-hard to find a schedule (or equivalently a permutation of jobs by Theorem 1) for which the first $f(n)$ elements of the sorted utility vector matches the first $f(n)$ elements of the sorted max-min fair utility vector. We state the result in the following theorem.

Theorem 2. *Given a set of machines $M = \{1, ..., m\}$ and a set of jobs $N = \{1, ..., n\}$, scheduling jobs to achieve $f(n)$-max-min utility optimal is NP-hard, for any increasing function $f(n)$ for which $f^{-1}(n)$ is polynomially bounded in n. The result holds even if all the utility functions are the same, i.e., $U_j(C_j) = U(C_j),\ \forall j \in N$.*

For instance, Theorem 2 holds for any sublinear function $f(n) = n^{\epsilon}$, for any $\epsilon \in (0, 1]$, but not for $f(n) = \log(n)$.

In the case of identical utility functions, $U_j(C_j) = U(C_j)$, $\forall j \in N$, it is easy to observe that "max-min" fairness among utilities is equivalent to "min-max" among the completion times. In the latter problem, we minimize the largest completion time across the jobs, and then successively minimize the next largest completion time as long as it does not affect the previous largest completion time, and so on. We formally state this fact in the following lemma.

Lemma 1. *In the case that $U_j(C_j) = U(C_j),\ \forall j \in N$, max-min fairness among utilities is equivalent to min-max of completion times.*

Proof. Given that the utility function $U(.)$ is not increasing, the result is immediate. □

Proof of Theorem 2. We prove the theorem for the special case when all jobs' utility functions are the same. This implies NP-hardness of the problem for general cases with any non-increasing utility functions. To prove this, we use a reduction from the *Minimum Vertex Cover Problem* which is known to be NP-hard [33]. An instance $\mathcal{I}$ of Minimum Vertex Cover Problem is given by a graph $G = (V, E)$, where the goal is to find a *minimum cardinality* set of vertices $W \subset V$ such that each edge $e \in E$ is incident to at least one vertex of W. We use $\mathrm{VC}(G)$ to denote the cardinality of W. We map this instance to an instance $\mathcal{I}'$ of the problem of $f(n)$-min-max completion times using a polynomial time procedure.

Instance $\mathcal{I}'$ has $m = |E| + n'$ machines, one for each edge $e \in E$, plus n' extra machines to be specified shortly, and $n = |V| + n'$ jobs, one for each vertex $v \in V$ and extra n' jobs. Let d_v denote the degree of vertex v in G. For each vertex $v \in V$, we consider a job $j(v)$ consisting of d_v tasks $(j(v), e)$, such that $p_{j(v)e} = 1$ if edge $e \in E$ is incident to v, and 0 otherwise. We refer to these jobs as *vertex jobs*. The remaining n' jobs, each has a unit-sized task on one of the last n' machines, such that each of the n' machines only has one task to process. We refer to these jobs as *dummy jobs*. We choose $n' = f^{-1}(|V|) - |V|$. Note that $f(|V|) \leq |V|$ and f is an increasing function (and so is f^{-1}), therefore, $n' \geq 0$. At the end of this construction, each machine has either 1 or 2 tasks to process; hence, all the jobs can be scheduled in two time slots. Consider a schedule with the following properties: (1) it finishes all the jobs using two time slots, (2) all the n' dummy jobs are completed in the first time slot Note that the set of tasks completed in the second time slot belong to a set of vertex jobs. This set of vertex jobs creates a vertex cover for G, because each of the first $|E|$ machines has to process some task from these jobs in the second time slot.

Note that by the choice of n',

$$f(n) = f(|V| + n') = f(f^{-1}(|V|)) = |V| > \mathrm{VC}(G). \quad (3)$$

Out of the first $f(n) = V$ jobs in the sorted completion time vector, some jobs have completion time equal to 2 and some jobs have completion time equal to 1. To find the $f(n)$-min-max vector, we therefore need to minimize the number of jobs completed in the second time slot, which is equivalent to finding the minimum vertex cover of G. Note that the remaining jobs correspond to an independent set in graph G, and hence all their tasks can be scheduled in the first time slot. However, it is NP-hard to find the minimum vertex cover of graph $G = (V, E)$ [33]. □

As a result of Theorem 2, we can conclude that the max-min fairness problem for single-task jobs considered in [12] (see Section I-A for more details) and the max-min fairness scheduling of coflows considered in [18] are both NP-hard problems, that were not shown before. The proof is based on reduction of our problem to these scheduling problems. The details are omitted due to space constraint.

Corollary 1. *The max-min fair scheduling problems considered in [12], [17], [18] are NP-hard.*

IV. Defining Approximation Solutions

In single-objective optimization, in case the problem is NP-hard, we try to find approximation algorithms, which run in polynomial time, and can return a solution with provable guarantee on its distance from the optimal solution (e.g., approximation ratio) [34]. However, the optimization problem in our setting is *not* a single-objective optimization, as we aim to optimize a vector of objective functions in the sense of max-min fairness. Consequently, given that finding the optimal vector solution to our problem is NP-hard (Theorem 2), it is not clear how to define the approximation algorithms in our setting. In this section, we describe two possible definitions for approximation solutions. We focus on the case that all jobs' utility functions are the same, i.e., $U_j(C_j) = U(C_j)$, $j \in N$. Recall that by Lemma 1, this is equivalent to the problem of min-max of completion times, which is still NP-hard by Theorem 2. In Section VI, we discuss possible extensions to unequal utility functions.

A. k-min-max fair approximation

A natural way of extending the idea of approximation ratio is through αn-min-max, for some $\alpha < 1$, based on $f(n) = \alpha n$ in Definition 3. We can attempt to find an approximate algorithm (schedule) such that the first αn elements of its corresponding sorted completion time vector matches the first αn elements of the sorted min-max vector. However, Theorem 2 implies that even finding such a schedule is NP-hard for any constant $\alpha > 0$. Therefore, we ask for less, and consider *finding the first k elements of the sorted optimal vector, for a fixed constant $k < n$.*

B. Single-objective approximation

The second approach could be to formulate a single-objective optimization whose optimal solution coincides with the min-max vector. We can then use this single-objective optimization to measure the quality of an approximation solution to the min-max problem. We describe one such formulation based on an integer program.

An Equivalent Integer Program (IP). We formulate an Integer Program based on minimization of the total weighted completion times of jobs. In traditional minimization of total weighted completion times [23]–[25], each job j has a positive fixed weight w_j and the objective is to minimize $\sum_{j\in N} w_j C_j$. The optimization that we consider here is different as the weights of jobs are not fixed in advance and depend on their positions in permutation. Formally, for any position $l \in \{1, 2, \ldots, n\}$ and any job $j \in N$, we define a binary variable x_{lj} which is 1 if *job j is the l-th job to complete in the schedule*, and 0 otherwise. In view of Definition 2, $x_{lj} = 1$ is equivalent to having $\pi(j) = l$, when $\pi_i(j) = \pi(j)$ for all $i \in M$. We refer to $\{x_{lj}\}$ as permutation variables. Each position $l \in \{1, 2, \ldots, n\}$ is assigned a non-negative weight w_l. Define $C^{(l)}$ to be the completion time of the l-th job completed in the schedule

$$
\begin{aligned}
(\mathbf{IP})\ \min\ & \sum_{l=1}^{n} w_l C^{(l)} && \text{(4a)}\\
& C^{(l)} \geq \sum_{s=1}^{l} \sum_{j\in N} p_{ij} x_{sj},\ i \in M,\ 1 \leq l \leq n && \text{(4b)}\\
& \sum_{l=1}^{n} x_{lj} = 1,\ j \in N && \text{(4c)}\\
& \sum_{j\in N} x_{lj} = 1,\ 1 \leq l \leq n && \text{(4d)}\\
& x_{lj} \in \{0, 1\},\ 1 \leq l \leq n,\ j \in N && \text{(4e)}
\end{aligned}
$$

Constraint (4b) is based on the definition of permutation variables and the fact that the completion time of the l-th job is greater than completion time of its task on any machine i. Constraints (4c) and (4d), capture the requirement that each job is assigned to a position, and each position is assigned to a job, respectively. Let $C^{\star(l)}$ denote the value of completion time of the l-th job in an optimal solution to (IP). Observe that by the minimization objective, for any job there is a machine for which Constraint (4b) turns to equality at the optimal solution. Let $i^\star$ denote the machine for which $C^{\star(l-1)} = \sum_{s=1}^{l-1} \sum_{j\in N} p_{i^\star j} x^\star_{sj}$. Then, as a result of Constraint (4b) on machine $i^\star$ for the l-th job we have,

$$
C^{\star(l)} \geq \sum_{s=1}^{l} \sum_{j\in N} p_{i^\star j} x^\star_{sj} \geq \sum_{s=1}^{l-1} \sum_{j\in N} p_{i^\star j} x^\star_{sj} = C^{\star(l-1)}.
$$

This implies that $C^{\star(1)} \leq \cdots \leq C^{\star(n)}$, i.e. the values of $C^{\star(l)}$ are consistent with our definition of jobs' positions $l = 1, \cdots, n$. However, since a job l with no task on a machine i is assumed to have a task with zero processing time on that machine, and Constraint (4b) is on all machines $i \in M$, the completion time of the job may be dominated by one of its zero-processing tasks. This can result in a larger value for $C^{\star(l)}$ than the *actual* completion time of the l-th job in the schedule according to (1). We need to show that $C^{\star(l)}$ is indeed the completion time of the l-th job in the schedule.

Lemma 2. *For any job h and its corresponding position l (i.e., $x^\star_{lh} = 1$) in an optimal solution to IP (4),*

$$
C_h = C^{\star(l)} = \sum_{s=1}^{l} \sum_{j\in N} p_{i^\star j} x^\star_{sj},\ \textit{for some } i^\star \in M_h.
$$

Therefore, $C^{\star(l)}$ is indeed the completion time of job h in the schedule corresponding to optimal permutation variables $x^\star_{lj}$ (or its corresponding job permutation $\pi^\star$).

The proof of Lemma 2 is based on a contradiction argument and optimality of $C^\star$. We omit the proof due to page limit.

Next, we show that by an appropriate choice of weights w_l, $l = 1, 2, \ldots, n$, we can force the optimal solution to IP (4) to coincide with the optimal min-max vector of completion times. Recall the definition of $\tau^{(N)}$ in (2). The following lemma states the result for non-trivial instances of min-max problem.

Lemma 3. *Let $w_0 = (\tau^{(N)})^n$ and assume that $\tau^{(N)} \geq 2$ and $n \geq 3$. The optimal solution to IP (4) is an optimal solution for min-max problem if we set $w_l = w_0^l$.*

Proof Overview. Consider an optimal solution $C^{\star(l)}$, $l = 1, \cdots, n$, to IP (4), and let $\tilde{C}^{(l)}$ be the completion time of the l-th job in a min-max solution. The proof is by contradiction. Suppose $\{C^{\star(l)}\}_{l=1}^n$ is not a min-max optimal solution. Then, it follows that there must exist some position l, $1 \leq l \leq n$, for which the following relation holds,

$$
\begin{aligned}
& C^{\star(l')} = \tilde{C}^{(l')}\ \forall l' > l,\\
& C^{\star(l')} > \tilde{C}^{(l)} \text{ for } l' = l,\\
& C^{\star(l')} < \tilde{C}^{(l')}\ \forall l' < l.
\end{aligned}
$$

We then proceed to show that by the choice of weights as in the lemma's statement, even if the completion time of the l-th job, $C^{\star(l)}$ is greater than $\tilde{C}^{(l)}$ by only *one* time unit, we get $\sum_{l'=1}^{l} w_{l'} C^{\star(l')} > \sum_{l'=1}^{l} w_{l'} \tilde{C}^{(l')}$, which contradicts the optimality of $\{C^{\star(l)}\}_{l=1}^n$ for IP (4). We omit the details due to page limit. □

Note that the total number of bits required to represent the weights in Lemma 3 is polynomially bounded in the problem input. Specifically, the number of bits required to represent the largest weight w_n is $O(n^2 \log \tau^{(N)})$, therefore we need at most $O(n^3 \log \tau^{(N)})$ bits to represent all the weights.

V. Approximation Algorithms for Equal Utility Functions

In this section, we consider the case where all jobs' utility functions are the same. Before presenting our scheduling algorithms, we describe a set of permutations that contains an optimal schedule. Recall that for each job $j \in N$, M_j denotes the set of machines for which $p_{ij} > 0$.

Lemma 4. *Consider the problem of finding the optimal min-max solution of jobs' completion times. For any two jobs h and k, 1) If $p_{i,h} \leq p_{i,k}, \forall i \in M_h \cap M_k$, then there is an optimal schedule that job h precedes job k in the permutation. 2) If $p_{i,h} = p_{i,k} = p, \forall i \in M_h \cap M_k$, then there is an optimal schedule that jobs h and k are adjacent in the permutation.*

The proof of Lemma 4 is based on an exchange argument and is omitted due to space constraint. We use Lemma 4 later in this section to augment the solution of an algorithm.

A. k-Max-Min Scheduling Algorithm

We aim to find a k-min-max fair schedule as defined in Section IV-A. This is equivalent to finding the last k jobs in the corresponding optimal permutation. Algorithm 1 gives a description of our algorithm. It is based on dynamic programming and starts by finding the last job and moves backward to find the last k jobs in the optimal permutation.

Algorithm 1 k-Max-Min Algorithm

1. If $k > 1$,
   - 1.1. compute the busy duration of each machine $i \in M$, given the job set N as $\tau_i^{(N)} = \sum_{j \in N} p_{ij}$.
   - 1.2. Compute the set of candidate jobs to be the last job to complete as $I_N = \arg\min_{j \in N} \max_{i \in M} (\tau_i^{(N)} - p_{ij})$.
   - 1.3. For each job $j \in I_N$, run Algorithm 1 for $N \leftarrow N \setminus \{j\}$, and $k \leftarrow k-1$ and denote the output permutation by π^j. Assign $\pi^j(j) = n$ for $j \in I_N$.
   - 1.4. Compare the output permutations $\{\pi^j\}$, and set π_1 to be the one whose corresponding completion time vector dominates the others in the sense of min-max fairness.
2. Else ($k = 0$), $\tilde{N} = \varnothing$, $\pi_1 = \varnothing$.

Let π_1 be the output of Algorithm 1, and $\tilde{N} = \{j \in N : \pi_1(j) = n, \cdots, n-k+1\}$. To schedule remaining jobs, we can compute a random permutation over remaining jobs $N \setminus \tilde{N}$, and modify it by exchanging jobs' positions according to Lemma 4 to get a permutation π_2. We can then use $\pi = [\pi_2, \pi_1]$ to schedule all jobs.

1) Correctness of Algorithm 1: Consider a machine i. The time that this machine needs to process all its associated tasks is given by $\tau_i^{(N)}$ as defined in line 1.1. Therefore, there exists a task (i, j) that completes at time $\tau_i^{(N)}$. Also, the completion time of the last job in any optimal schedule is equal to $\tau^{(N)} = \max_{i \in M} \tau_i^{(N)}$, which is the optimal makespan (2). Now the algorithm needs to decide which job should it actually complete last in the schedule. Assume that it chooses job j as the last job to complete (equivalently, $\pi(j) = n$), then the second-largest completion time across all the jobs will be equal to

$$\tau^{(N \setminus \{j\})} = \max_{i \in M} \tau_i^{(N \setminus \{j\})} = \max_{i \in M} (\tau_i^{(N)} - p_{ij}).$$

Hence, the algorithm finds the set of jobs I_N such that $\tau^{(N \setminus \{j\})}$ is minimized for $j \in I_N$. Note that this is necessary in order to achieve a min-max fair vector. Also, note that the maximization in line 1.2 of the algorithm is over the set M and not M_j, for all $j \in N$, to ensure that position n is assigned to a job with the largest completion time. Applying a similar argument, we conclude that Algorithm 1 correctly finds the last k jobs in the optimal schedule.

2) Time Complexity of Algorithm 1: Observe that the size of set I_N (line 1.2) is at most n. This implies that running time of the algorithm is $O(kmn^k)$ which is polynomial in input size for a fixed value of k. If we set $k = n$, we need to check all the $n!$ possible permutations to find out the optimal solution. As we can observe from execution of Algorithm 1, the reason that we need to consider all possibilities for the optimal permutation of jobs (that can blow to $n!$) is that size of candidate set I_N is generally greater than one. Hence, the Algorithm requires to check which candidate job it should choose for each position. In the case that there is a unique candidate job at each iteration, the optimal permutation can be computed in $O(mn^2 + mn \log(p))$ time, where p is the maximum task processing time.

B. Perturbation-Based Scheduling Algorithm

Algorithm 2 Perturbation-Based Algorithm

1. Choose a constant $\epsilon > 0$.
2. For every job $j \in N$, draw a number ϵ_j randomly from interval $[0, \epsilon]$. Then update its tasks' processing times $p_{ij} \leftarrow p_{ij} + \epsilon_j$.
3. For $l = n$ to 1, compute the busy duration of each machine $i \in M$ corresponding to set N, as in line 1.1 of Algorithm 1.
4. Let $I_N = \arg\min_{j \in N} \max_{i \in M} (\tau_i^{(N)} - p_{ij})$.
5. If $|I_N| \neq 1$, go to line 2. Else, set the l-th position in the permutation to be the unique job $j^\star \in I_N$, i.e., $\pi(j^\star) = l$, and update $N \leftarrow N \setminus \{j^\star\}$.
6. Schedule jobs (with the original processing times) according to the obtained permutation π.

Algorithm 2 gives a description of our perturbation-based algorithm to schedule multi-task jobs so as to approximate the min-max completion time vector, in the single-objective approximation sense (Section IV-B). At a high level, given an instance of the problem, we perturb the tasks' processing times with a small random noise. This is an attempt to ensure in execution of Algorithm 1, the number of candidate jobs calculated in line 2 reduces to 1 with high probability. For each job j, we draw a noise ϵ_j uniformly at random from interval $[0, \epsilon]$. Define $p'_{ij} = p_{ij} + \epsilon_j$ to be the processing times in the perturbed instance. Similar to Algorithm 1, for the perturbed instance, we compute the optimal permutation starting from the last position n. The perturbation noises in practice are not real numbers, hence, the probability that the set of candidate jobs for the l-th position, $l = 1, \ldots, n$, contains more than one job is small but not zero. To resolve possible collisions in a candidate set, we have lines 5 in Algorithm 2.

1) Evaluation of Algorithm 2: Consider an instance of our problem. Let π denote the permutation of jobs computed by Algorithm 2. We use optimal objective value of IP (4) to measure the distance of the computed solution by Algorithm 2 to the optimal solution.

Theorem 3. *Let π be the permutation of jobs computed by Algorithm 2 and $\mathcal{C}$ denote the objective value of IP (4) according to this permutation. Also let OPT be the objective value of IP for an optimal min-max fair schedule. Then, $\mathcal{C} \leq OPT + g(\epsilon)$, where $g(\epsilon)$ is a strictly decreasing function in ϵ, and $g(\epsilon) \to 0$ as $\epsilon \to 0$.*

We refer to the instance before applying perturbation as *original instance*. Recall that optimal solution of IP (4) is equivalent to the optimal max-min solution for the original instance; therefore, difference of the two objective values $\mathcal{C}$ and OPT, denoted by $g(\epsilon)$ is a sound metric to evaluate the quality of permutation π computed by Algorithm 2 for the original instance. Moreover, note that we can choose any ϵ by considering sufficiently large number of bits to represent the perturbation noise which incurs greater complexity. This issue is addressed in Subsection V-B2.

Proof Overview. The permutation π computed by Algorithm 2 is optimal for the perturbed instance. Therefore, by Lemma 3, π yields to the smallest objective value, $\widetilde{\text{OPT}}$, for IP (4) (when equipped with weights that correspond to the perturbed instance). Next, we apply the optimal permutation of the original instance, $\pi^\star$, on the perturbed instance and use $\bar{\mathcal{C}}$ to denote its IP's value. We find the relationship between OPT and $\mathcal{C}$, by comparing their values with $\widetilde{\text{OPT}}$ and $\bar{\mathcal{C}}$. It follows that $g(\epsilon) = (n^2+1)\epsilon \sum_{l=1}^{n} l w_l + \epsilon^2 f(\epsilon)$, for a polynomial function f. We omit the details due to space constraint. □

2) Time Complexity of Algorithm 2: Let b denote the number of bits used to represent the perturbation noises. The probability of having more than one job in set I_N in the first iteration is less than $\binom{n}{2} \times 2^{-b}$ by the union bound. Therefore, the probability of not encountering any collision in I_N is at least $1 - 2^{-(b+1)}n^2$. Choosing $b = 3 \times \log(n)$, the average number of times we should execute the algorithm to pass the first iteration successfully is less than $\frac{2n}{2n-1} \leq 2$. Applying the same argument, the average number of times needed to successfully complete all the iterations is polynomial in the input size. Therefore, Algorithm 2, on average, has polynomial time complexity in the input size of the original instance (i.e., $O(mn^2 + mn\log(p))$). In simulations for Google trace (Section VII), the algorithm always found each position successfully in one try.

VI. General Utility Functions

The main obstacle in extending the results in Section IV and V to unequal utility functions is that the jobs' positions in the optimal permutation, based on jobs' completion times, may not be the same as the jobs' positions according to jobs' utilities. Algorithm 1 used the fact that for any set of jobs N, there exists a job that completes at the optimal makespan $\tau^{(N)}$ (Equation (2)). This gives the min-max of completion times and also helps us decide which job to schedule last. However, in the case of unequal utility functions, the job that is scheduled last with the largest completion time may not be the job with the worst utility. Therefore, Algorithm 1 cannot be generalized to find the last k jobs with the worst utilities in the case of general utility functions.

Nevertheless, we present a generalization of the perturbation-based algorithm (Algorithm 2) to unequal utility functions. Since utility functions are assumed to be Lipschitz continuous (bounded first derivative), we can choose the noise parameter ϵ small enough such that job utilities do not change dramatically after perturbing task processing times. The algorithm is essentially the same as Algorithm 2, except that we do not update processing times in line 2, and instead in line 4, the set of candidate jobs is computed as

$$I_N = \arg\max_{j \in N} \min_{i \in M_j} U_j(\tau_i^{(N)} + \epsilon_j).$$

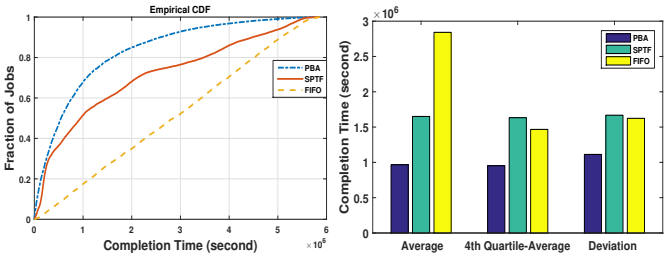
Note that the positions of jobs in the obtained permutation π by this algorithm, is neither the same as the positions based on the sorted completion time vector (Definition 2), nor the same as the positions based on the sorted utility vector. Nevertheless, we can use this permutation π to schedule jobs. We evaluate the performance of this algorithm empirically in simulations.

VII. Simulation Results

In this section, we evaluate the performance of our algorithms using a real traffic trace from a large Google cluster [35]. The original trace is based on ∼11000 servers over a month long period. In our experiments, we filter jobs and consider a set of jobs with at most 200 number of tasks which are about 99% of all the jobs in the *production* class. Also, in order to have reasonable traffic density on each machine (since otherwise the problem is trivial), we consider a cluster with 200 machines and randomly map machines of the original set to machines of this set. In simulations, we choose parameter ϵ in Algorithm 2 and its generalized version to be 10^{-4} times the smallest task processing time in the data set. For brevity, in Figures, we refer to both Algorithm 2 and its generalized version as PBA (*Perturbation-Based Algorithm*).

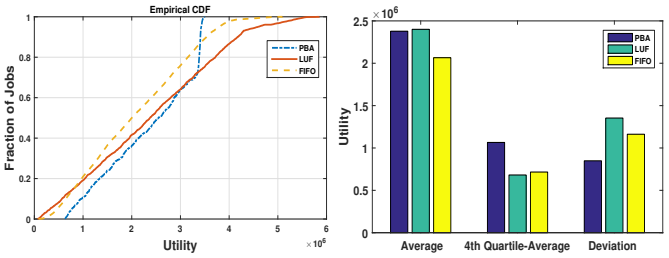
We evaluate the performance of our algorithms in two cases:

- **Equal Utility Functions:** When all the jobs have the same utility function, lexicographic max-min of utilities is equivalent to lexicographic min-max of completion times (by Lemma 1). We compare Algorithm 2 (PBA) with *First-In First-Out* (FIFO), and *Shortest Processing Time First* (SPTF). In FIFO, we list jobs based to their arrival times and schedule tasks on each machine according to this list. In SPTF, we list tasks on each machine in non-increasing order of their processing times, and schedule tasks starting from the first task in this list.
- **General Utility Functions:** We consider linear utility functions for jobs with different slopes that capture the priority information which is available for each job in the data set. In this case, we compare the performance of generalized Algorithm 2 as described in Section VI (PBA), *First-In*



(a) Empirical CDF. (b) Average and Deviation.

Fig. 1: Job *completion times* under PBA, SPTF, and FIFO in the offline setting. Lower average and lower deviation is better.



(a) Empirical CDF. (b) Average and Deviation.

Fig. 2: Job *utilities* under PBA, LUF, and FIFO, in the offline setting. Higher averages and lower deviation is better.

First-Out (FIFO), and *Largest Utility First* (LUF). In LUF, we consider a utility for each task, using the utility function of its corresponding job. Then on each machine at any time, we list tasks according to their utility values, and schedule the task that gives the largest utility upon completion, then move to the next task, and so on.

We examine algorithms by looking at *Cumulative Distribution Function* (CDF) for job completion times and utilities, in online and offline setting, with equal and unequal utility functions. In addition, we report 3 performance metrics:

- *Average*: the average of completion times of jobs (in the case of equal utility functions), or the average of their utilities (in the case of unequal utility functions).
- *4th Quartile-Average*: the average of the worst 25% of completion times or utilities among jobs. This metric indicates how much each algorithm starves long or low-utility jobs compared to the average.
- *Deviation*: the standard deviation of the job completion times (or their utilities) from the average, which is a metric of overall fairness to all jobs

In the case of equal utility functions, we report the results for job completion times, hence, smaller average and smaller 4th quartile-average are preferable. In the case of general utilities, we report the results for job utility values, hence in this case, larger average and larger 4th quartile-average are preferable. Moreover, in both cases, smaller deviation value for an algorithm shows that it has a better overall fairness.

A. Offline Setting

We first present the results in the offline setting.

1) Equal Utility Functions: Figure 1a depicts the empirical CDF of PBA, SPTF, and FIFO. Furthermore, Figure 1b shows the three aforementioned performance metrics (Average, 4th Quartile-Average, and Deviation) for job completion times. Not only our algorithm is better in terms of fairness, as shown by its deviation which is 0.65 of deviation of the other algorithms, and does not starve long jobs compared to other algorithms, but interestingly it also improves the average job completion time by a factor of almost 1.7 and 3, compared to SPTF and FIFO, respectively.

2) General Utility Functions: In the data set, each jobs has a priority that roughly represents how sensitive it is to latency. There are 9 different values of job priorities. For job j, we consider the utility function $U_j(t) = P_j \times (\tau - t)$, where τ is the makespan of completing all the jobs (a constant just to ensure utilities are positive) and P_j is the priority of job j.

Figure 2a shows the empirical CDF of PBA, LUF, and FIFO, and Figure 2b shows the average, 4th quartile-average, and deviation of jobs' obtained utilities. The worst utility among all the jobs under PBA is 9.5 and 6.9 times greater than the worst utility under LUF and FIFO, respectively. Note that, the CDF plot of PBA is sharper around its average value. PBA reduces deviation in obtained utilities, compared to LUF and FIFO, by a factor of 1.6 and 1.4, respectively, while it achieves almost the same average utility as LUF.

B. Online Setting

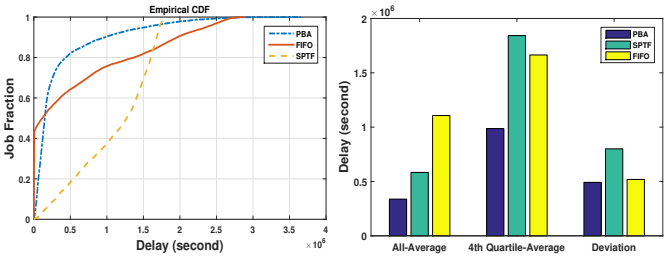
In the online setting, jobs arrive according to the arrival times information in the data set. Upon arrival of a job, SPTF updates its list and proceeds with the new list. However, it does not preempt an ongoing task in a machine. Similar to SPTF, LUF updates its list upon arrival of a job and proceeds with the new list in a non-preemptive fashion.

To extend our algorithm to online setting, we choose a parameter δ that is tunable. We divide time into time intervals of length δ time-units. At the beginning of each interval, we run the offline algorithm on the set of jobs consisting of jobs that are not scheduled yet and those that arrived in the previous interval. Further, tasks on the boundary of intervals are processed non-preemptively, i.e., if some task is running in some machine according to the previously computed schedule, we let it finish and then proceed with the new schedule. It is preferred to start with a smaller value of δ at the beginning, to avoid delaying the initial jobs in the system for δ amount of time before starting scheduling them. Therefore, we use an adaptive choice of δ to improve the performance of our online algorithm. We choose the length of the i-th interval, δ_i, as

$$\delta_i = \delta_0/(1 + \alpha \times \exp(-\beta i)),\ i = 1, 2, \cdots$$

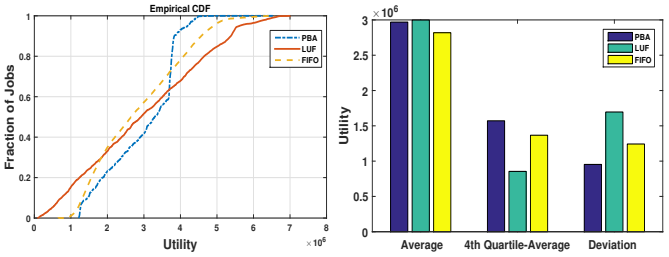
We choose $\delta_0 = 3.3 \times 10^5$ seconds, and $\alpha = 50$ and $\beta = 3$. All the jobs arrive over a time horizon of 3.3×10^6 seconds.

1) Equal Utility Functions: Figure 3a and 3b show the performance of PBA, SPTF, and FIFO in the online setting. We present the results in terms of job delay, which is the time between a job arrival and its completion time. PBA improves the average job delay by a factor of 1.7 and 3.3, compared to SPTF and FIFO. It also achieves better fairness by a factor



(a) Empirical CDF. (b) Average and Deviation.

Fig. 3: Job *delays* under PBA, SPTF, and FIFO, in the online setting. Lower averages and lower deviation is better.



(a) Empirical CDF. (b) Average and Deviation.

Fig. 4: Job *utilities* under PBA, LUF, and FIFO, in the online setting. Higher averages and lower deviation is better.

1.9 and 1.7 compared to SPTF and FIFO for the 4th quartile-average.

2) General Utility Functions: In the online setting, variable t used in the job utility function is measured from arrival of job j to the system. Figure 4a shows the empirical CDF of PBA, SPTF, and FIFO. Further, Figure 4b shows the average and deviation of jobs' obtained utilities. The smallest utility value among all the jobs under PBA is 1.9 and 14.6 times greater than the smallest utility value of jobs under FIFO and LUF, respectively. PBA also improves utility deviation compared to LUF and FIFO by a factor of 1.8 and 1.3, respectively.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, 2007, pp. 59–72.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, 2010.

[4] 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 *Proc. of ACM Symposium on Cloud Computing*, 2012, p. 7.

[5] 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.

[6] W. Wang, B. Liang, and B. Li, "Multi-resource fair allocation in heterogeneous cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 10, pp. 2822–2835, 2015.

[7] A. Chandra, W. Gong, and P. Shenoy, "Dynamic resource allocation for shared data centers using online measurements," in *International Workshop on Quality of Service*. Springer, 2003, pp. 381–398.

[8] "Apache hadoop yarn," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2018.

[9] "Hadoop fair scheduler," http://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html, 2018.

[10] "Hadoop capacity scheduler," https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, 2018.

[11] L. D. Briceno, B. Khemka, H. J. Siegel, A. A. Maciejewski, C. Groër, G. Koenig, G. Okonski, and S. Poole, "Time utility functions for modeling and evaluating resource allocations in a heterogeneous computing system," in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 2011, pp. 7–19.

[12] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Need for speed: Cora scheduler for optimizing completion-times in the cloud," in *IEEE INFOCOM*, 2015, pp. 891–899.

[13] J. Jaffe, "Bottleneck flow control," *IEEE Transactions on Communications*, vol. 29, no. 7, pp. 954–962, 1981.

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

[15] B. Avi-Itzhak and H. Levy, "On measuring fairness in queues," *Advances in applied probability*, vol. 36, no. 3, pp. 919–936, 2004.

[16] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM, 2012, pp. 31–36.

[17] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. Tsang, "Rush: A robust scheduler to manage uncertain completion-times in shared clouds," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 242–251.

[18] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *INFOCOM 2016*, 2016, pp. 1–9.

[19] D. E. Irwin, L. E. Grit, and J. S. Chase, "Balancing risk and reward in a market-based task service," in *International Symposium on High-Performance Distributed Computing,*, 2004, pp. 160–169.

[20] 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.

[21] S. Dimopoulos, C. Krintz, and R. Wolski, "Justice: A deadline-aware, fair-share resource allocator for implementing multi-analytics," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 233–244.

[22] R. Ahmadi, U. Bagchi, and T. A. Roemer, "Coordinated scheduling of customer orders for quick response," *Naval Research Logistics (NRL)*, vol. 52, no. 6, pp. 493–512, 2005.

[23] N. Garg, A. Kumar, and V. Pandit, "Order scheduling models: Hardness and algorithms," in *Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*. Springer, 2007, pp. 96–107.

[24] 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.

[25] 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.

[26] D. P. Palomar and M. Chiang, "A tutorial on decomposition methods for network utility maximization," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 8, pp. 1439–1451, 2006.

[27] P.-L. Yu, "Domination structures and nondominated solutions," in *Multiple-Criteria Decision Making*. Springer, 1985, pp. 163–214.

[28] J. P. Evans and R. E. Steuer, "A revised simplex method for linear multiple objective programs," *Mathematical Programming*, vol. 5, no. 1, pp. 54–72, 1973.

[29] H. D. Sherali, "Equivalent weights for lexicographic multi-objective programs: Characterizations and computations," *European Journal of Operational Research*, vol. 11, no. 4, pp. 367–379, 1982.

[30] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2015.

[31] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 21–21.

[32] M. Ehrgott, *Multicriteria optimization*. Springer Science & Business Media, 2005, vol. 491.

[33] M. R. Garey and D. S. Johnson, *Computers and intractability*. W. H. Freeman New York, 2002, vol. 29.

[34] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2013.

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