This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

Scheduling Jobs across Geo-Distributed Datacenters with Max-Min Fairness

Li Chen, Shuhao Liu, Baochun Li, *Fellow, IEEE,* and Bo Li, *Fellow, IEEE*

Abstract—It has become routine for large volumes of data to be generated, stored, and processed across geographically distributed datacenters. To run a single data analytic job on such geo-distributed data, recent research proposed to distribute its tasks across datacenters, considering both data locality and network bandwidth across datacenters. Yet, it remains an open problem in the more general case, where *multiple* analytic jobs need to *fairly* share the resources at these geo-distributed datacenters. In this paper, we focus on the problem of assigning tasks belonging to multiple jobs across datacenters, with the specific objective of achieving *max-min fairness* across jobs sharing these datacenters, in terms of their job completion times. We formulate this problem as a lexicographical minimization problem, which is challenging to solve in practice due to its inherent multi-objective and discrete nature. To address these challenges, we iteratively solve its single-objective subproblems, which can be transformed to equivalent linear programming (LP) problems to be efficiently solved, thanks to their favorable properties. As a highlight of this paper, we have designed and implemented our proposed solution as a fair job scheduler based on Apache Spark, a modern data processing framework. With extensive evaluations of our real-world implementation on Amazon EC2 and large-scale simulations, we have shown convincing evidence that max-min fairness has been achieved and the worst job completion time has been significantly improved using our new job scheduler.

Index Terms—Geo-distributed Datacenter Networks, Wide-Area Big Data Analytics, Scheduling, Fairness

1 Introduction

It is increasingly common for large volumes of data to be generated and processed in a geographically distributed fashion, across multiple datacenters around the world. Popular data analytic frameworks, such as MapReduce [1] and Spark [2], are extensively employed to process such large volumes of data efficiently. A data analytic *job* typically proceeds in consecutive computation *stages*, each of which consisting of a number of computation *tasks* that are executed in parallel. To start a new computation stage, intermediate data from the preceding stage needs to be fetched, which may initiate multiple network flows.

When input data is located across multiple datacenters, a naive approach is to gather all the data to be processed locally within a single datacenter. Naturally, transferring huge amounts of data across datacenters may be slow and inefficient, since bandwidth on inter-datacenter network links is limited [3]. Existing research (*e.g.*, [4], [5]) has shown that better performance can be achieved if tasks in an analytic job can be distributed across datacenters, and located closer to the data to be processed. In this case, designing the best possible task assignment strategy to assign tasks to datacenters is important, since different strategies lead to different flow patterns across datacenters, and ultimately, different job completion times.

When designing optimal task assignment strategies, however, existing works in the literature [4], [5] only considered a single data analytic job. The problem of assigning tasks belonging to multiple jobs across datacenters remains open. Given the limited amount of resources at each datacenter, multiple jobs are inherently competing for resources with each other. It is, therefore, important to maintain *fairness* when allocating such a shared pool of resources, which cannot be achieved if tasks from one job are assigned without considering the other jobs.

In this paper, we propose a new task assignment strategy that is designed to achieve *max-min fairness* across multiple jobs with respect to their performance, as they compete for the limited pool of shared resources across multiple geo-distributed datacenters. To be more specific, we wish to minimize the job completion times across all concurrent jobs, while maintaining max-min fairness. Such a problem can be formally formulated as a *lexicographical minimization* problem, which has unique challenges that make it difficult to solve this problem with multiple objectives. The task assignment problem is essentially an integer optimization problem, which in general is NP-hard [6].

To address these challenges, we first consider the subproblem of minimizing the worst (longest) job completion time among all the concurrent jobs, which turns out to have a *totally unimodular* coefficient matrix for linear constraints, based on an in-depth investigation of the problem structure. Such a nice property guar-

- *Li Chen, Shuhao Liu and Baochun Li are with the Department of Electrical and Computer Engineering, University of Toronto, Canada. E-mail: {lchen,shuhao,bli}@ece.utoronto.ca. Bo Li is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China. E-mail: {bli}@cse.ust.hk*

- *The research is supported in part by the NSERC Collaborative Research and Development Grant, and by grants from RGC GRF under the contracts 16211715 and 16206417, a grant from RGC CRF under the contract C7036-15G.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

antees that the extreme points in a feasible solution polyhedron are integers. Moreover, with several steps of non-trivial transformations, we show that the optimal solution to the original problem can be obtained by solving an equivalent problem with a *separable convex objective*. With these structures identified, we can then apply the λ-technique and linear relaxation to obtain a linear programming (LP) problem, which is guaranteed to have the same solution to the original problem. As a result, any LP solver can be used for minimizing the completion time of each job, and to efficiently compute the overall assignment decisions that achieve the optimal completion times with max-min fairness.

To demonstrate the practicality of our proposed solution, we have designed and implemented a new job scheduler to assign tasks from multiple jobs to geo-distributed datacenters, in the context of Apache Spark. Our experimental results on multiple Amazon EC2 datacenters have shown that our new scheduler is effective in optimizing job completion times and achieving max-min fairness.

Highlights of our original contributions are as follows. *First*, as motivated by our example in Sec. 2, we focus on jointly assigning tasks from multiple data parallel jobs across multiple datacenters, which considers the interplay between these jobs when sharing a limited pool of datacenter resources. *Second*, our problem, formulated as a lexicographical minimization problem in Sec. 3, has both the discrete and multi-objective nature that make it challenging to solve. Fortunately, with a careful investigation of its structure, we are able to identify the favorable properties of totally unimodular constraints and a separable convex objective, thus transforming it into an equivalent LP problem to be efficiently and elegantly solved (Sec. 4 and Sec. 5). *Finally*, to show its effectiveness and practicality, we have completed both large-scale simulations and a real-world implementation of our proposed solution within the Spark job scheduling framework, and conducted extensive evaluations across multiple datacenters in Amazon EC2 (Sec. 6).

2 Background and Motivation

In this section, we first present an overview of the execution of a data analytic job whose input data is stored across geographically distributed datacenters. We then consider the sharing of resources in these datacenters across multiple concurrent jobs, and provide a motivating example to illustrate the need for fair job scheduling.

2.1 Data Analytic Jobs in the Wide Area

It is typical for a data analytic job to contain tens or hundreds of tasks, supported by a data parallel framework, such as MapReduce and Spark. These tasks are parallel to or dependent upon each other, and network flows are generated between dependent tasks, since tasks in the subsequent stage need to fetch intermediate data from tasks in the current stage. For example, in a MapReduce job, a set of `map` tasks are first launched to read input data partitions and generate intermediate results; then `reduce` tasks would fetch such intermediate data from `map` tasks for further processing, which involves transferring data over the network.

In the case of running tasks in a data analytic job across multiple datacenters, data may be transferred over inter-datacenter links, which may become bottlenecks due to their limited bandwidth availability. Our design objective in this paper is to compute the best way to assign tasks belonging to *multiple* jobs to geo-distributed datacenters, so that all jobs can achieve their best possible performance with respect to their completion times, without harming the performance of others. This implies that *max-min fairness* needs to be achieved across jobs sharing the datacenters, in terms of their job completion times.

For a better intuition of our problem, we use Fig. 1 to show an example with two data analytic jobs sharing three geo-distributed datacenters. For job `A`, both of its tasks, `tA1` and `tA2`, require 100 MB of data from input dataset `A1` stored in `DC1`, and 200 MB of data from `A2` located at `DC3`. For job `B`, the amounts of data to be read by task `tB1` from dataset `B1` in `DC2` and `B2` in `DC3` are both 200 MB; while task `tB2` needs to read 200 MB of data from `B1` and 300 MB from `B2`. These tasks are to be assigned to available computing slots in the three datacenters, each with two slots, two slots, and one slot, respectively. The amounts of available bandwidth of inter-datacenter links are illustrated in the figure, with the unit of MB/s.

For each job, a different assignment of its tasks will lead to flows of different sizes traversing different links, thus resulting in different job completion times. Moreover, both jobs must share and compete for the same pool of computing resources across these datacenters. For example, since `DC3` only has one available computing slot, if we assign a task from one job, tasks from the other job cannot be assigned. In order to achieve the best possible performance for both jobs, we need to consider the placement of their tasks jointly, rather than independently.

2.2 An Intuition on Task Assignment

We have illustrated two ways of assigning tasks to datacenters in Fig. 2 and Fig. 3. Intuitively, `DC3` is a favorable location for tasks from both jobs, since they all have part of their input data stored in this datacenter, and the links of `DC1`-`DC3` and `DC2`-`DC3` both have high bandwidth. For simplicity, we assume that all the tasks are identical, with the same execution time, thus the job completion time is determined by the network transfer time. If the scheduler tries to optimize task assignment of these jobs independently, the result is shown in Fig.2. To optimize the assignment of job `A`, task `tA2` would be assigned to the only available computing slot in `DC3`, and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

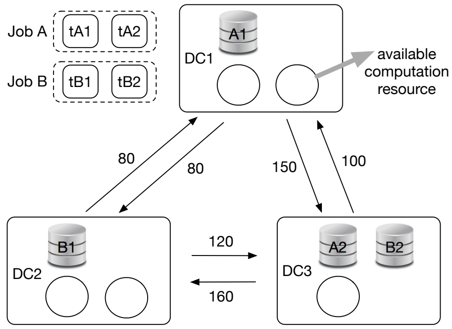


Fig. 1: An example of scheduling multiple jobs fairly across geo-distributed datacenters.

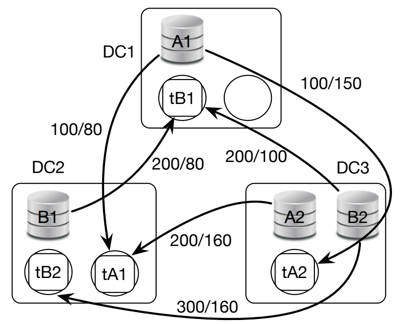


Fig. 2: A task assignment that favors the performance of job `A` at the cost of job `B`.

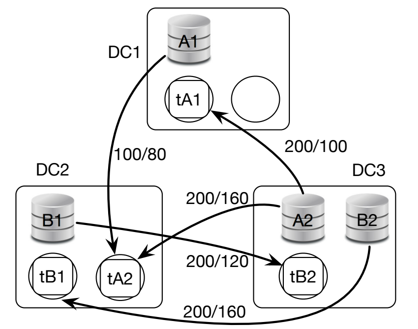


Fig. 3: The optimal assignment for both job `A` and job `B`.

`tA1` would be placed in `DC2`, which result in a network transfer time of $\max\{100/80, 200/160, 100/150\} = 1.25$ seconds. Then, if the scheduler continues to optimize the assignment of job `B`, `DC1` and `DC2` would be selected to distribute task `tB1` and `tB2`, respectively, resulting in the transfer time of $\max\{200/80, 200/100, 300/160\} = 2.5$ seconds for job `B`.

However, this placement is not optimal when considering the performance of these jobs jointly. Instead, we show the optimal assignment for both jobs satisfying max-min fairness in Fig. 3. With this assignment, task `tB2` of job `B` would occupy the computing slot in `DC3`, which avoids the transfer of 300 MB data from dataset `B2`. Task `tB1` is assigned to `DC2` rather than `DC1`, which takes advantage of the high bandwidth of the `DC3`-`DC2` link. As a result, the flow patterns are illustrated in the figure. In this assignment, the network transfer times of job `A` and `B` are $\max\{200/100, 100/80, 200/160\} = 2$ seconds, and $\max\{200/160, 200/120\} = 5/3$ seconds, respectively. Compared with the independent assignment in Fig. 2 where the worst performance is 2.5 seconds, this assignment results in the worst network transfer time of 2 seconds (job `A`), which is optimal if we wish to minimize the worst job completion time, and is fair in terms of the performance achieved by both jobs.

We are now ready to formally construct a mathematical model to study the problem of optimizing task assignment with max-min fairness across multiple jobs to be achieved.

3 Model and Formulation

We consider a set of data parallel jobs $\mathcal{K} = \{1, 2, \cdots, K\}$ submitted to the scheduler for task assignment. The input data of these jobs are distributed across a set of geo-distributed datacenters, represented by $\mathcal{D} = \{1, 2, \cdots, J\}$. Each job $k \in \mathcal{K}$ has a set of parallel tasks $\mathcal{T}_k = \{1, 2, \cdots, n_k\}$ to be launched on available computing slots in these datacenters. We use a_j to denote the capacity of available computing slots in datacenter $j \in \mathcal{D}$.

For each task $i \in \mathcal{T}_k$ of job k, the time it takes to complete consists of both the network transfer time, denoted by $c^k_{i,j}$, to fetch the input data if the task is assigned to datacenter j, and the execution time represented by $e^k_{i,j}$. The network transfer time is determined by both the amount of data to be read, and the bandwidth on the link the data traverses. Let S^k_i denote the set of datacenters where the input data of task i from job k are stored, called the source datacenters of this task for convenience. The task needs to read the input data from each of its source datacenters $s \in S^k_i$, the amount of which is represented by $d^{k,s}_i$. Let $b_{s,j}$ [1] represent the bandwidth of the link from datacenter s to datacenter j ($s \neq j$). Hence, the network transfer time of task $i \in \mathcal{T}_k$, if assigned to datacenter j, is expressed as follows:

$$c^k_{i,j} = \begin{cases} 0, & \text{when } S^k_i = \{j\}; \\ \max\limits_{s \in S^k_i, s \neq j} d^{k,s}_i / b_{sj}, & \text{otherwise.} \end{cases} \tag{1}$$

This indicates that when all the input data of task i are stored in the same datacenter j, *i.e.*, $S^k_i = \{j\}$, there would not be any traffic generated across datacenters, and thus the network transfer time is 0. Otherwise, task i fetches data from each of its remote source datacenter $s \in S^k_i, s \neq j$ with the completion time of $d^{k,s}_i / b_{sj}$. As the network transfer completes when input data from all the source datacenters have been fetched, the transfer time $c^k_{i,j}$ is represented as the maximum of $d^{k,s}_i / b_{sj}$ over all the remote source datacenters.

The assignment of a task is represented with a binary variable $x^k_{i,j}$, indicating whether the i-th task of job k is assigned to datacenter j. A job k completes when its slowest task finishes, thus the job completion time of k, represented by τ_k, is determined by the maximum completion time among all of its tasks, expressed as follows:

$$\tau_k = \max_{i \in \mathcal{T}_k, j \in \mathcal{D}} x^k_{i,j} (c^k_{i,j} + e^k_{i,j}) \tag{2}$$

As the computing slots in all the datacenters are shared by tasks from multiple jobs, we would like to obtain an optimal task assignment without exceeding the resource capacities. To be more specific, our scheduler would decide the assignment of all the tasks, aiming to

1. On popular cloud platforms (e.g., Amazon EC2 and Google Could), inter-datacenter wide-area networks are provided as a shared service, where user-generated flows will compete with millions of other flows. As a result, each inter-datacenter TCP flow will get a fair share of the link capacity. Our measurement with iperf3 on EC2 verifies this assumption.

optimize the worst performance achieved among all the jobs with respect to their job completion times, and then optimize the next worst performance without impacting the previous one, and so on. This is executed repeatedly until the completion times have been optimized for all the jobs. Such an objective can be rigorously formulated as a *lexicographical minimization* problem, with the following definitions as its basis.

Definition 1: Let $\langle \boldsymbol{v} \rangle_k$ denote the k-th ($1 \leq k \leq K$) largest element of $\boldsymbol{v} \in \mathbb{Z}^K$, implying $\langle \boldsymbol{v} \rangle_1 \geq \langle \boldsymbol{v} \rangle_2 \geq \cdots \geq \langle \boldsymbol{v} \rangle_K$. Intuitively, $\langle \boldsymbol{v} \rangle = (\langle \boldsymbol{v} \rangle_1, \langle \boldsymbol{v} \rangle_2, \cdots, \langle \boldsymbol{v} \rangle_K)$ represents the non-increasingly sorted version of $\boldsymbol{v}$.

Definition 2: For any $\boldsymbol{\alpha} \in \mathbb{Z}^K$ and $\boldsymbol{\beta} \in \mathbb{Z}^K$, if $\langle \boldsymbol{\alpha} \rangle_1 < \langle \boldsymbol{\beta} \rangle_1$ or $\exists k \in \{2, 3, \cdots, K\}$ such that $\langle \boldsymbol{\alpha} \rangle_k < \langle \boldsymbol{\beta} \rangle_k$ and $\langle \boldsymbol{\alpha} \rangle_i = \langle \boldsymbol{\beta} \rangle_i, \forall i \in \{1, \cdots, k-1\}$, then $\boldsymbol{\alpha}$ is *lexicographically smaller* than $\boldsymbol{\beta}$, represented as $\boldsymbol{\alpha} \prec \boldsymbol{\beta}$. Similarly, if $\langle \boldsymbol{\alpha} \rangle_k = \langle \boldsymbol{\beta} \rangle_k, \forall k \in \{1, 2, \cdots, K\}$ or $\boldsymbol{\alpha} \prec \boldsymbol{\beta}$, then $\boldsymbol{\alpha}$ is *lexicographically no greater* than $\boldsymbol{\beta}$, represented as $\boldsymbol{\alpha} \preceq \boldsymbol{\beta}$.

Definition 3: $\operatorname{lexmin}_{\boldsymbol{x}} \boldsymbol{f}(\boldsymbol{x})$ represents the *lexicographical minimization* of the vector $\boldsymbol{f} \in \mathbb{R}^N$, which consists of N objective functions of $\boldsymbol{x}$. To be particular, the optimal solution $\boldsymbol{x}^* \in \mathbb{R}^K$ achieves the optimal $\boldsymbol{f}^*$, in the sense that $\boldsymbol{f}^* = \boldsymbol{f}(\boldsymbol{x}^*) \preceq \boldsymbol{f}(\boldsymbol{x}), \forall \boldsymbol{x} \in \mathbb{R}^K$.

With these definitions, we are now ready to formulate our optimal task assignment problem among sharing jobs as follows:

$$\begin{aligned} \operatorname*{lexmin}_{\boldsymbol{x}} \quad & \boldsymbol{f} = (\tau_1, \tau_2, \cdots, \tau_K) & (3) \\ \text{s.t.} \quad & \tau_k = \max_{i \in \mathcal{T}_k, j \in \mathcal{D}} x_{i,j}^k (c_{i,j}^k + e_{i,j}^k), \forall k \in \mathcal{K} & (4) \\ & \sum_{k \in \mathcal{K}} \sum_{i \in \mathcal{T}_k} x_{i,j}^k \leq a_j, \quad \forall j \in \mathcal{D} & (5) \\ & \sum_{j \in \mathcal{D}} x_{i,j}^k = 1, \quad \forall i \in \mathcal{T}_k, \ \forall k \in \mathcal{K} & (6) \\ & x_{i,j}^k \in \{0, 1\}, \quad \forall i \in \mathcal{T}_k, \ \forall j \in \mathcal{D}, \ \forall k \in \mathcal{K} & (7) \end{aligned}$$

where constraint (4) represents the completion time of each job k as aforementioned. Constraint (5) indicates that the total number of tasks to be assigned to datacenter j does not exceed its capacity a_j, which is the total number of available computing slots. Constraint (6) implies that each task should be assigned to a single datacenter.

The objective is a vector $\boldsymbol{f} \in \mathbb{R}^K$ with K elements, each standing for the completion time of a particular job $k \in \mathcal{K}$. According to the previous definitions, the optimal $\boldsymbol{f}^*$ is lexicographically no greater than any $\boldsymbol{f}$ obtained with a feasible assignment, which means that when sorting them in a non-increasing order, if their k-th largest element satisfies $\langle \boldsymbol{f}^* \rangle_{k'} = \langle \boldsymbol{f} \rangle_{k'}, \forall k' < k$ and $\langle \boldsymbol{f}^* \rangle_k \neq \langle \boldsymbol{f} \rangle_k$, then we have $\langle \boldsymbol{f}^* \rangle_k < \langle \boldsymbol{f} \rangle_k$. This implies that the first largest element of $\boldsymbol{f}^*$, *i.e.*, the slowest completion time, is the minimum among all $\boldsymbol{f}$. Then among all $\boldsymbol{f}$ with the same worst completion time, the second worst completion time in $\boldsymbol{f}^*$ is the minimum, and so on. In this way, solving this problem would result in an optimal assignment vector $\boldsymbol{x}^*$, with which all the job completion times are minimized.

4 Optimizing the Worst Completion Time among Concurrent Jobs

Problem (3) is a vector optimization with multiple objectives. In this section, we consider the single-objective subproblem of optimizing the worst job performance as follows:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \max_{k \in \mathcal{K}} (\tau_k) & (8) \\ \text{s.t.} \quad & \text{Constraints } (4), (5), (6) \text{ and } (7). \end{aligned}$$

which is a primary step towards solving the original problem, to be elaborated in the next section.

Substituting the completion time τ_k in the objective with the expression in constraint (4), we have the following problem with the non-linear constraint (4) eliminated:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \max_{k \in \mathcal{K}} \big(\max_{i \in \mathcal{T}_k, j \in \mathcal{D}} x_{i,j}^k (c_{i,j}^k + e_{i,j}^k) \big) & (9) \\ \text{s.t.} \quad & \text{Constraints } (5), (6) \text{ and } (7). \end{aligned}$$

Though this problem is an integer programming problem, we will show that it can be transformed into an equivalent linear programming (LP) problem after an in-depth investigation of its structure. As a result of such a transformation, it can be solved efficiently to obtain the optimal schedule vector $\boldsymbol{x}$. Our transformation takes advantage of its features of *separable convex objective* and *totally unimodular linear constraints*, and it involves three major steps to be elaborated in the following subsections.

4.1 Separable Convex Objective

In the first step, we will show that the optimal solution for Problem (9) can be obtained by solving a problem with a separable convex objective function, which is represented as a summation of convex functions with respect to each single variable $x_{i,j}^k$.

We first show that the optimal solution of Problem (9) can be obtained by solving the following problem:

$$\begin{aligned} \operatorname*{lexmin}_{\boldsymbol{x}} \quad & \boldsymbol{g} = (\phi(x_{1,1}^1), \cdots, \phi(x_{i,j}^k), \cdots, \phi(x_{n_K,J}^K)) \\ \text{s.t.} \quad & \text{Constraints } (5), (6) \text{ and } (7). \end{aligned}$$

where $\phi(x_{i,j}^k) = x_{i,j}^k (c_{i,j}^k + e_{i,j}^k), \forall i \in \mathcal{T}_k, \ \forall j \in \mathcal{D}, \ \forall k \in \mathcal{K}$, and $\boldsymbol{g}$ is a vector with the dimension of $M = |\boldsymbol{g}| = J \sum_{k=1}^K n_k$. For this problem, the objective includes minimizing the maximum element in $\boldsymbol{g}$, which is the worst completion time across all the jobs. Therefore, the optimal assignment variables $\boldsymbol{x}^*$ that gives $\boldsymbol{g}^*$ is also the optimal solution for Problem (9).

Let $\varphi(\boldsymbol{g})$ define a function of $\boldsymbol{g}$:

$$\varphi(\boldsymbol{g}) = \sum_{m=1}^{|\boldsymbol{g}|} |\boldsymbol{g}|^{g_m} = \sum_{m=1}^{M} M^{g_m}$$

where g_m is the m-th element of $\boldsymbol{g}$.

Lemma 1: *$\varphi(\cdot)$ preserves the order of lexicographically no greater ($\preceq$), i.e., $\boldsymbol{g}(\boldsymbol{x}^*) \preceq \boldsymbol{g}(\boldsymbol{x}) \iff \varphi(\boldsymbol{g}(\boldsymbol{x}^*)) \leq \varphi(\boldsymbol{g}(\boldsymbol{x}))$.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

Proof: We first consider $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{Z}^K$ that satisfies $\boldsymbol{\alpha} \prec \boldsymbol{\beta}$. If we use the integer $\tilde{k}(1 \le \tilde{k} \le K)$ to represent the first non-zero element of $\langle\boldsymbol{\alpha}\rangle - \langle\boldsymbol{\beta}\rangle$, we have $\langle\boldsymbol{\alpha}\rangle_k = \langle\boldsymbol{\beta}\rangle_k, \forall \tilde{k} \le k \le K$ and $\langle\boldsymbol{\alpha}\rangle_{\tilde{k}} < \langle\boldsymbol{\beta}\rangle_{\tilde{k}}$. Assume $\langle\boldsymbol{\alpha}\rangle_{\tilde{k}} = m$, then $\langle\boldsymbol{\beta}\rangle_{\tilde{k}} \ge m+1$.

$$\begin{aligned}
\varphi(\boldsymbol{\alpha}) &= \sum_{k=1}^{K} K^{\langle\boldsymbol{\alpha}\rangle_k} = \sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\alpha}\rangle_k} + K^{\langle\boldsymbol{\alpha}\rangle_{\tilde{k}}} + \sum_{k=\tilde{k}+1}^{K} K^{\langle\boldsymbol{\alpha}\rangle_k} \\
&\le \textstyle\sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\alpha}\rangle_k} + K^{\langle\boldsymbol{\alpha}\rangle_{\tilde{k}}} + (K-\tilde{k})K^{\langle\boldsymbol{\alpha}\rangle_{\tilde{k}}} \\
&= \textstyle\sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\alpha}\rangle_k} + (K+1-\tilde{k}) \cdot K^{\langle\boldsymbol{\alpha}\rangle_{\tilde{k}}} \\
&< \textstyle\sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\alpha}\rangle_k} + K \cdot K^m,
\end{aligned}$$

where the first inequality holds as $\langle\boldsymbol{\alpha}\rangle_{\tilde{k}} \ge \langle\boldsymbol{\alpha}\rangle_k, \forall \tilde{k}+1 \le k \le K$.

$$\begin{aligned}
\varphi(\boldsymbol{\beta}) &= \sum_{k=1}^{K} K^{\langle\boldsymbol{\beta}\rangle_k} = \sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\beta}\rangle_k} + K^{\langle\boldsymbol{\beta}\rangle_{\tilde{k}}} + \sum_{k=\tilde{k}+1}^{K} K^{\langle\boldsymbol{\beta}\rangle_k} \\
&> \textstyle\sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\beta}\rangle_k} + K^{\langle\boldsymbol{\beta}\rangle_{\tilde{k}}} + (K-\tilde{k}) \cdot 0 \\
&\ge \textstyle\sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\beta}\rangle_k} + K \cdot K^m.
\end{aligned}$$

Given that $\sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\alpha}\rangle_k} = \sum_{k=1}^{\tilde{k}-1} K^{\langle\boldsymbol{\beta}\rangle_k}$, we have proved that $\varphi(\boldsymbol{\alpha}) < \varphi(\boldsymbol{\beta})$.

If $\boldsymbol{\alpha} = \boldsymbol{\beta}$, which means that $\langle\boldsymbol{\alpha}\rangle_k = \langle\boldsymbol{\beta}\rangle_k, \forall 1 \le k \le K$, it is trivially true that $\varphi(\boldsymbol{\alpha}) = \sum_{k=1}^{K} K^{\langle\boldsymbol{\alpha}\rangle_k} = \sum_{k=1}^{K} K^{\langle\boldsymbol{\beta}\rangle_k} = \varphi(\boldsymbol{\beta})$. Thus, we have proved $\boldsymbol{\alpha} \preceq \boldsymbol{\beta} \Longrightarrow \varphi(\boldsymbol{\alpha}) \le \varphi(\boldsymbol{\beta})$.

We further prove $\varphi(\boldsymbol{\alpha}) \le \varphi(\boldsymbol{\beta}) \Longrightarrow \boldsymbol{\alpha} \preceq \boldsymbol{\beta}$ by proving its contrapositive: $\neg(\boldsymbol{\alpha} \preceq \boldsymbol{\beta}) \Longrightarrow \varphi(\boldsymbol{\alpha}) > \varphi(\boldsymbol{\beta})$. $\neg(\boldsymbol{\alpha} \preceq \boldsymbol{\beta})$ implies $\boldsymbol{\alpha} \ne \boldsymbol{\beta}$ and the first non-zero element of $\langle\boldsymbol{\alpha}\rangle - \langle\boldsymbol{\beta}\rangle$ is positive, which further indicates the first non-zero element of $\langle\boldsymbol{\beta}\rangle - \langle\boldsymbol{\alpha}\rangle$ is negative, *i.e.*, $\boldsymbol{\beta} \prec \boldsymbol{\alpha}$. Thus, the contrapositive is equivalent to $\boldsymbol{\beta} \prec \boldsymbol{\alpha} \Longrightarrow \varphi(\boldsymbol{\beta}) < \varphi(\boldsymbol{\alpha})$, which has already been proved previously using the exchanged notations of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.

With $\boldsymbol{\alpha} \preceq \boldsymbol{\beta} \iff \varphi(\boldsymbol{\alpha}) \le \varphi(\boldsymbol{\beta})$ holding for any $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ of the same dimension, we complete the proof by letting $\boldsymbol{\alpha} = \boldsymbol{g}(\boldsymbol{x}^*)$ and $\boldsymbol{\beta} = \boldsymbol{g}(\boldsymbol{x})$. □

Based on Lemma 1, we have

$$\operatorname*{lexmin}_{\boldsymbol{x}} \ \boldsymbol{g} \iff \min_{\boldsymbol{x}} \ \ \varphi(\boldsymbol{g}) = \sum_{k\in\mathcal{K}} \sum_{i\in\mathcal{T}_k} \sum_{j\in\mathcal{D}} M^{\phi(x_{i,j}^k)}$$

where the objective function $\varphi(\boldsymbol{g})$ is a summation of the term $M^{\phi(x_{i,j}^k)}$, which is a convex function of the single variable $x_{i,j}^k$.

Therefore, solving Problem (9) is equivalent to solving the following problem with a separable convex objective:

$$\begin{aligned}
\min_{\boldsymbol{x}} \quad & \sum_{k\in\mathcal{K}} \sum_{i\in\mathcal{T}_k} \sum_{j\in\mathcal{D}} M^{\phi(x_{i,j}^k)} \qquad (10)\\
\text{s.t.} \quad & \text{Constraints } (5), (6) \text{ and } (7).
\end{aligned}$$

4.2 Totally Unimodular Linear Constraints

In the second step, we investigate the coefficient matrix of linear constraints (5) and (6). An m-by-n matrix is *totally unimodular* [6], if it satisfies two conditions: 1) any of its elements belongs to $\{-1, 0, 1\}$; 2) any row subset $R \subset \{1, 2, \cdots, m\}$ can be divided into two disjoint sets, R_1 and R_2, such that $|\sum_{i\in R_1} a_{ij} - \sum_{i\in R_2} a_{ij}| \le 1, \forall j \in \{1, 2, \cdots, n\}$.

***Lemma** 2:* The coefficients of constraints (5) and (6) form a totally unimodular matrix.

Proof: Let $\mathbf{A}_{m\times n}$ denote the coefficient matrix of all the linear constraints (5) and (6), where $m = J + \sum_{k=1}^{K} n_k$, representing the total number of the constraints, and $n = J\sum_{k=1}^{K} n_k$, denoting the dimension of the variable $\boldsymbol{x}$.

It is obvious that any element of $\mathbf{A}_{m\times n}$ is either 0 or 1, satisfying the first condition. For any row subset $R \subset \{1, 2, \cdots, m\}$, we can select all the elements that belong to $\{1, 2, \cdots, J\}$ to form the set R_1. As such, R is divided into two disjoint sets, R_1 and $R_2 = R - R_1$. It is easy to check that for coefficient matrix of constraint (5), the summation of all its rows, represented by rows $\{1, 2, \cdots, J\}$, is a $1 \times n$ vector with all the elements equal to 1. Similarly, for coefficient matrix of constraint (6), the summation of all its rows, represented by rows $\{J+1, J+2, \cdots, J+\sum_{k=1}^{K} n_k\}$, is also a $1 \times n$ vector whose elements are 1. Hence, we can easily derive that $\sum_{i\in R_1} a_{ij} \le 1, \sum_{i\in R_2} a_{ij} \le 1, \forall j \in \{1, 2, \cdots, n\}$. Eventually, we have $|\sum_{i\in R_1} a_{ij} - \sum_{i\in R_2} a_{ij}| \le 1, \forall j \in \{1, 2, \cdots, n\}$, and the second condition is satisfied.

In summary, we have shown that both conditions for total unimodularity are satisfied, thus the coefficient matrix $\mathbf{A}_{m\times n}$ is totally unimodular. □

4.3 Structure-Inspired Equivalent LP Transformation

In the final step, exploiting the problem structure of totally unimodular constraints and separable convex objective, we can use the λ-representation technique [7] to transform Problem (10) to a linear programming problem that has the same optimal solution.

For a single integer variable $y \in \mathcal{Y} = \{0, 1, \cdots, Y\}$, the convex function $h : \mathcal{Y} \to \mathbb{R}$ can be linearized with the λ-representation as follows:

$$\begin{aligned}
&h(y) = \sum_{s\in\mathcal{Y}} h(s)\lambda_s, \quad y = \sum_{s\in\mathcal{Y}} s\lambda_s \\
&\sum_{s\in\mathcal{Y}} \lambda_s = 1, \quad \lambda_s \in \mathbb{R}^+, \ \forall s \in \mathcal{Y}.
\end{aligned}$$

In our problem, we apply the λ-representation technique to each convex function $h_{i,j}^k(x_{i,j}^k) = M^{\phi(x_{i,j}^k)} : \{0, 1\} \to \mathbb{R}$ as follows:

$$h_{i,j}^k(x_{i,j}^k) = \sum_{s\in\{0,1\}} M^{s(c_{i,j}^k + e_{i,j}^k)} \lambda_{i,j}^{k,s} = \lambda_{i,j}^{k,0} + M^{c_{i,j}^k + e_{i,j}^k} \lambda_{i,j}^{k,1}$$

which removes the variable $x_{i,j}^k$ by sampling at each of its possible value $s \in \{0, 1\}$, weighted by the newly

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

introduced variables $\lambda_{i,j}^{k,s} \in \mathbb{R}^+, \forall s \in \{0,1\}$ that satisfy

$$x_{i,j}^k = \sum_{s\in\{0,1\}} s\lambda_{i,j}^{k,s} = \lambda_{i,j}^{k,1}$$
$$\sum_{s\in\{0,1\}} \lambda_{i,j}^{k,s} = \lambda_{i,j}^{k,0} + \lambda_{i,j}^{k,1} = 1$$

Further, with linear relaxation on the integer constraints (7), we obtain the following linear programming problem:

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{\lambda}} \quad & \sum_{k\in\mathcal{K}}\sum_{i\in\mathcal{T}_k}\sum_{j\in\mathcal{D}} (\lambda_{i,j}^{k,0} + M^{c_{i,j}^k+e_{i,j}^k}\lambda_{i,j}^{k,1}) \qquad (11) \\ \text{s.t.} \quad & x_{i,j}^k = \lambda_{i,j}^{k,1}, \quad \forall k \in \mathcal{K}, i \in \mathcal{T}_k, j \in \mathcal{D} \\ & \lambda_{i,j}^{k,0} + \lambda_{i,j}^{k,1} = 1, \quad \forall k \in \mathcal{K}, i \in \mathcal{T}_k, j \in \mathcal{D} \\ & \lambda_{i,j}^{k,0}, \lambda_{i,j}^{k,1}, x_{i,j}^k \in \mathbb{R}^+, \ \forall k \in \mathcal{K}, i \in \mathcal{T}_k, j \in \mathcal{D} \\ & \text{Constraints (5) and (6).} \end{aligned}$$

Theorem 1: An optimal solution to Problem (11) is an optimal solution to Problem (8).

Proof: The property of total unimodularity ensures that an optimal solution to the relaxed LP problem (11) has integer values of $x_{i,j}^k$, which is an optimal solution to Problem (10), and thus an optimal solution to Problem (9) as demonstrated in Sec. 4-A. Moreover, Problem (8) and (9) are equivalent forms, completing the proof. □

Therefore, the optimal assignment that minimizes the worst completion time among all the jobs can be obtained by solving Problem (11) with efficient LP solvers, such as MOSEK [8].

5 Iteratively Optimizing Worst Completion Times to Achieve Max-Min Fairness

With the subproblem of minimizing the worst completion time efficiently solved as an LP problem (11), we continue to solve our original multi-objective problem (3) by minimizing the next worst completion time repeatedly.

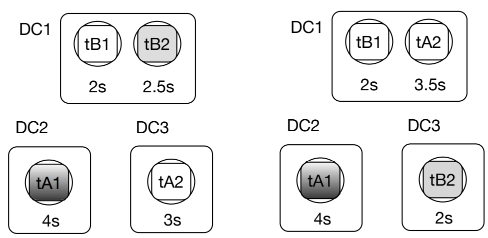
After solving the subproblem, it is known that the optimal worst completion time is achieved by job k^*, whose slowest task i^* is assigned to datacenter j^*. We then fix the computed assignment of the slowest task of job k^*, which means that the corresponding schedule variable $x_{i^*,j^*}^{k^*}$ is removed from the variable set $\boldsymbol{x}$ for the next round. Also, since task i^* is to be assigned to datacenter j^*, it is intuitive that all the assignment variables associated with it should be fixed as zero and removed from $\boldsymbol{x}$: $x_{i^*,j}^{k^*} = 0, \forall j \neq j^*, j \in \mathcal{D}$.

As we have fixed a part of the assignment, the resource capacities should be updated in our problem constraints in the next round. For example, if $x_{i^*,j^*}^{k^*} = 1$, which means that the i^*th task of job k^* would be assigned to datacenter j^*, then for the problem in the next round, $x_{i^*,j^*}^{k^*}$ is no longer a variable. The resource capacity constraints should be updated as $\sum_{k\in\mathcal{K}}\sum_{i\in\mathcal{T}_k,(k,i)\neq(k^*,i^*)} x_{i,j^*}^k \leq a_{j^*} - 1$.

Algorithm 1: Performance-Optimal Task Assignment among Jobs with Max-Min Fairness.

Input:
  Input data sizes $d_i^{k,s}$ and link bandwidth b_{sj} to obtain network transfer time $c_{i,j}^k$ (by Eq. 1); execution time $e_{i,j}^k$; datacenter resource capacity a_j;
Output:
  Task assignment $x_{i,j}^k, \forall k \in \mathcal{K}, \forall i \in \mathcal{T}_k, \forall j \in \mathcal{D}$;

Initialize $\mathcal{K}' = \mathcal{K}$;
while $\mathcal{K}' \neq \varnothing$ **do**
  Solve the LP Problem (11) to obtain the solution $\boldsymbol{x}$;
  Obtain $x_{k^*,i^*}^{j^*} = \operatorname{argmax}_{x_{i,j}^k \in \boldsymbol{x}} \phi(x_{i,j}^k)$;
  Fix $x_{i^*,j}^{k^*}, \forall j \in \mathcal{D}$; remove them from variable set $\boldsymbol{x}$;
  Update the corresponding resource capacities in Constraints (5);
  Set $\phi(x_{i,j}^{k^*}) = x_{i,j}^{k^*}(c_{i^*,j^*}^{k^*} + e_{i^*,j^*}^{k^*}), \ \forall i \in \mathcal{T}_{k^*}, \forall j \in \mathcal{D}$;
  Remove k^* from $\mathcal{K}'$;
end while



(a) (b)

Fig. 4: Two possible assignments of tasks from two jobs.

Moreover, the completion time of job k^* is obtained as $\phi(x_{i^*,j^*}^{k^*})$, the completion time of its slowest task i^* assigned to datacenter j^*, yet the assignment of other tasks has not been fixed, which would be the variables $(x_{i,j}^{k^*}, \forall(i,j) \neq (i^*,j^*))$ of the problem in the next round. We set the associated completion times of these variables as $x_{i,j}^{k^*}(c_{i^*,j^*}^{k^*} + e_{i^*,j^*}^{k^*})$. The rationale is that no matter how fast other tasks of k^* complete, the completion time is determined by the slowest task i^*. This ensures that the completion time optimized in the next round is achieved by another job, rather than a task of job k^* (other than its slowest). This is better illustrated with the example in Fig. 4.

In this example, after the calculation in the first round, task `tA1` assigned in `DC2` achieves the worst completion time of 4s, among the two sharing jobs. In the second round, if we do not change the associated completion time for another task `tA2`, assignment (a) would be calculated as the optimal solution, since it achieves the completion times of $(4, 3, 2.5, 2)$ for the four tasks, which is lexicographically smaller than $(4, 3.5, 2, 2)$ achieved by assignment (b). However, assignment (a) minimizes the next worst completion time for another task of the same job `A`, which does not match our original objective to optimize the worst completion time for job `B`. Instead, if

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

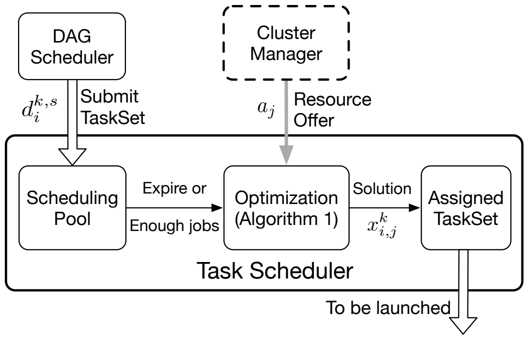


Fig. 5: The implementation of our task assignment algorithm in Spark.

we set the associated completion times of all job `A`'s tasks as 4, the objective function achieved by assignment (b) would be $(4, 4, 2, 2)$, better than $(4, 4, 2.5, 2)$ given by assignment (a). In this way, the optimization can correctly choose assignment (b) to achieve the optimal completion time of job `B`. Note that although the completion time of task `tA2` in assignment (b) is longer than in assignment (a), the completion time of job `A` remains the same, which is 4s.

As a result, the subproblem in the next round is solved over a decreased set of variables with updated constraints and objectives, so that the next worst job completion time would be optimized, without impacting the worst job performance in this round. Such a procedure is repeatedly executed until the last worst completion time of jobs has been optimized, and the max-min fairness has been achieved, as summarized in Algorithm 1.

6 Performance Evaluation

Having proved the theoretical optimality and efficiency of our scheduling solution, we proceed to implement it in Apache Spark, and demonstrate its effectiveness in optimizing job completion times in real-world experiments. Moreover, we present an extensive array of large-scale simulation results to demonstrate the effectiveness of our algorithm in achieving max-min fair job performance.

6.1 Real-World Implementation and Experiments

6.1.1 Design and Implementation

In Apache Spark [2], a job can be represented by a Directed Acyclic Graph (DAG), where each node represents a task and each directed edge indicates a precedence constraint. In general, the problem of assigning all the tasks in a DAG to a number of worker nodes, with the objective of minimizing the job completion time, is known as NP-Complete [9]. As a practical and efficient design, Spark schedules tasks stage by stage, which is handled by the *DAG scheduler*. When a job is submitted, it is transformed into a DAG of tasks, categorized into a set of stages. The DAG scheduler will then submit the tasks within each stage, called a `TaskSet`, to the task scheduler whenever the stage is ready to be scheduled, implying that all its parent stages have completed.

Fortunately, the task scheduler in Spark has access to most of the information that our algorithm needs as its input. As a result, we have implemented our new task assignment algorithm as an extension to Spark's `TaskScheduler` module. The design of our implementation is illustrated in Fig. 5. In our implementation, as soon as a `TaskSet` in a job has been submitted by the DAG scheduler to the task scheduler, they will be immediately queued in the scheduling pool. With a number of concurrent jobs waiting to be scheduled, our algorithm will be triggered when a preset timer expires or when the number of pending jobs in the pool exceeds a certain threshold.

To optimize the assignment of tasks, our algorithm requires knowledge about the size and location of the output data from each `map` task, represented by $d_i^{k,s}$ in our formulation. Such knowledge can be obtained from the `MapOutputTracker`, which are further saved in the `TaskSet` of `reduce` tasks. The available bandwidth (b_{sj}) between each pair of datacenters (s to j) can be measured with the `iperf2` utility. The task execution time $e_{i,j}^k$ can be obtained from historical data, which is the common practice for recurrent jobs. (We can also utilize advanced prediction techniques such as [10] to estimate the task execution time in our future work.) In addition, information about the amount of available resources, corresponding to a_j in our formulation, can be obtained from the cluster manager. Now that all the input required by our algorithm is ready, an optimal assignment will be computed for all the tasks in the scheduling pool, through iteratively formulating and solving updated versions of linear programming problems, solved by the LP solver in the `Breeze` optimization library [11].

After the assignment has been computed, it will be recorded in the corresponding `TaskSet`, overriding the original task assignment preferences. When the tasks are finally submitted for execution, these assignment preferences will be satisfied in a greedy manner. Since the scheduling decisions will satisfy resource constraints by considering *Resource Offers*, each task in the `TaskSet` can be launched in any available computing slot in the assigned datacenter.

6.1.2 Experimental Setup

We are now ready to evaluate our real-world implementation with an extensive set of experiments deployed across 6 datacenters in Amazon EC2, located in a geographically distributed fashion across different continents. The available bandwidth between each pair of datacenters, measured with the `iperf2` utility and averaged over 5 measurements over the period of 20 minutes, is shown in Table 1. Compared to the intra-datacenter network where the available bandwidth is around 1 Gbps, bandwidth on inter-datacenter links are much more limited: almost all inter-continental links have less than 100 Mbps of available bandwidth. This

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

TABLE 1: Available bandwidth across geo-distributed datacenters (Mbps).

	Virginia	Oregon	Ireland	Sing	Sydney	SP
Virginia	1000	169	154	52	53	104
Oregon	-	1000	71	69	77	68
Ireland	-	-	1000	49	40	65
Singapore	-	-	-	1000	58	35
Sydney	-	-	-	-	1000	38
San Paulo	-	-	-	-	-	1000

"Sing" is short for "Singapore", and "SP" is short for "San Paulo".

confirms the observation that transferring large volumes of data across datacenters is likely to be time-consuming.

In our experiments, we have used a total of 12 on-demand Virtual Machine (VM) instances as Spark workers in our Spark cluster, located across 6 datacenters. Two special VM instances in Virginia (`us-east-1`) have been used as the Spark master node and the Hadoop File System (HDFS) [12] Name Node, respectively. All instances are of type `m3.large`, each with 2 vCPUs, 7.5 GB of memory, and a 32GB Solid-State Drive. In each instance, we run Ubuntu Server 14.04 LTS 64-bit (HVM), with Java 1.8 and Scala 2.11.8 installed. Hadoop 2.6.4 is installed to provide HDFS support for Spark. Our own implementation of the task assignment algorithm is based on Spark 1.6.1, a recent release as of July 2016. Our Spark cluster runs in the standalone mode, with all configurations left as default. No external resource manager (*e.g.,* YARN) or database system is activated.

In order to illustrate the efficiency of our task assignment algorithm, we use the legacy `Sort` application as the benchmark workload. We choose this workload because it is simple but primitive. As one of the simplest MapReduce applications, `Sort` has only one map and one reduce stage. However, its `sortByKey()` operation is a basic building block for many complex data analytics applications, especially in Spark SQL. It triggers an all-to-all shuffle, which introduces heavier cross-node traffic than other reduce operations such as `reduceByKey()`.

In our experiments, we have implemented a `Sort` application with multiple jobs, and submitted it to Spark for execution. The jobs are submitted to Spark in parallel threads, triggering concurrent jobs to share the resources in the cluster. Therefore, the task assignment decisions for these concurrent jobs will be made and enforced by our implementation in the `TaskScheduler`. To evaluate the performance under different workloads, we run the workload with 3, 4 and 5 concurrent jobs as separate experiments.

For each `Sort` job in our benchmark application, the default parallelism is set to 3. In other words, the job will trigger 3 reduce tasks to sort the input dataset, which has 3 partitions distributed on 3 randomly chosen worker nodes. The input dataset is prepared as a step in the `map` task. Each partition of the generated dataset is 100 MB in size, containing 10,000 key-value pairs. Then, as the start of the `reduce` task, these key-values will be shuffled over the network. Since each datacenter has only two workers, a fraction of the shuffled traffic will be sent over inter-datacenter network links, which are likely to be the performance bottleneck. Our task assignment algorithm is specifically designed to mitigate the negative effects of such bottlenecks.

6.1.3 Experimental Results

We conducted three groups of experiments, with 3, 4, and 5 concurrent jobs, respectively. In the first two groups, each job has three tasks; while in the third group, the total number of tasks of the five jobs is set as the total number of available computing slots, which is 12 in our Spark cluster. In each experiment, the job completion times achieved with our optimal task assignment algorithm is compared with that achieved with the default scheduling in Spark, used as the baseline in our comparison study.

The results of our experiments are presented in Fig. 6 and Fig. 7, showing the worst completion times and the second worst completion times among concurrent jobs, respectively. With respect to the worst completion time, it is easy to see that our algorithm always performs better than the baseline in Spark, with a performance improvement of up to 66%, as shown in Fig. 6. With respect to the second worst completion time, our algorithm does not theoretically guarantee that it is smaller than that achieved with any unfair placement. Even without guarantees, our algorithm always shows better performance than the baseline in Spark. These experiments have shown convincing evidence that our algorithm — optimizing for max-min fairness — is effective in maximizing the worst completion time and achieving best possible performance for all concurrent jobs.

To offer a more in-depth examination and show why such a performance improvement can be achieved, we consider the 7-th run in the second group of our experiment, with the sharing relationship and bandwidth shown in Fig. 8. The six circles represent the six datacenters used in our experiment. `A1` is located in the Virginia datacenter, representing the data required by tasks from job `A`. For each task in job `A` to be scheduled, a fraction of data needs to be read from all the three datasets (`A1`, `A2` and `A3`). As each datacenter has two available computing slots, the assignment of 12 tasks from four jobs becomes a one-to-one mapping. In such a limited resource scenario, the assignment of a task is tightly coupled with each other. It becomes more difficult for the default strategy in Spark to find a good assignment, without optimizing across all the tasks. Moreover, the wide range of available bandwidth between datacenters is not taken into consideration by the default scheduling in Spark, which also explains the performance improvement of our strategy over the baseline.

Furthermore, to evaluate the practicality of our algorithm, we have recorded the time it takes to calculate optimal solutions. Fig. 9 illustrates the computation times, each averaged over 10 runs, with the number of variables varying from 12 to 120. The linear program in our algorithm is efficient, as it takes about 1 second

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

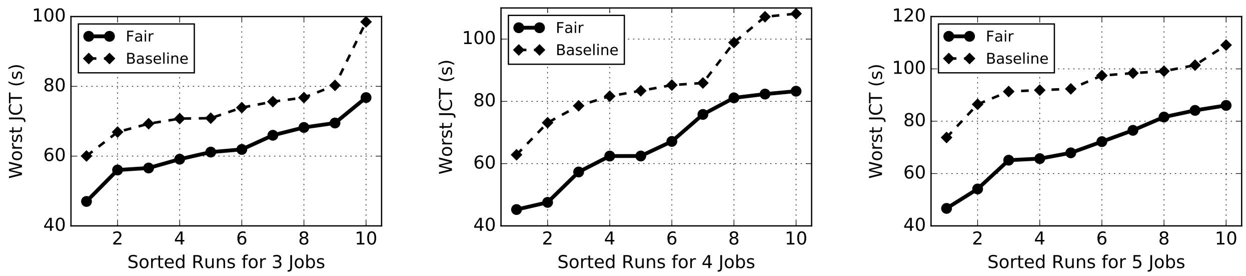


Fig. 6: The worst job completion time in a set of concurrent jobs.

Fig. 7: The second-worst job completion time in a set of concurrent jobs.

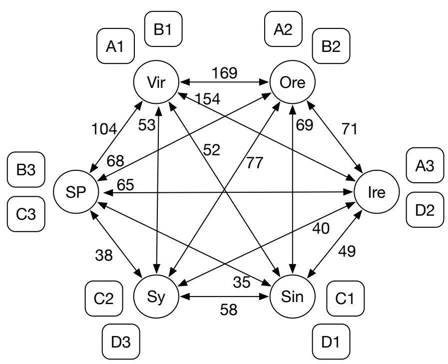


Fig. 8: The location of input data for four jobs across six datacenters.

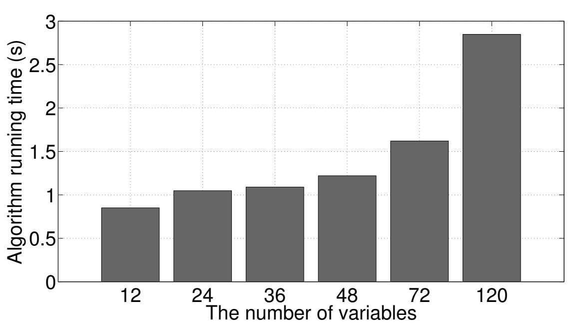


Fig. 9: The computation times of our algorithm at different scales.

to obtain the solution for 48 variables. The computation time is less than 3 seconds for 120 variables, which is acceptable compared with the transfer times across datacenters that could be tens or hundreds of seconds. In our experiment, the algorithm is running in the VM with 2 vCPUs. We envision that with more powerful servers for the scheduler in a production environment, the running time could be even smaller.

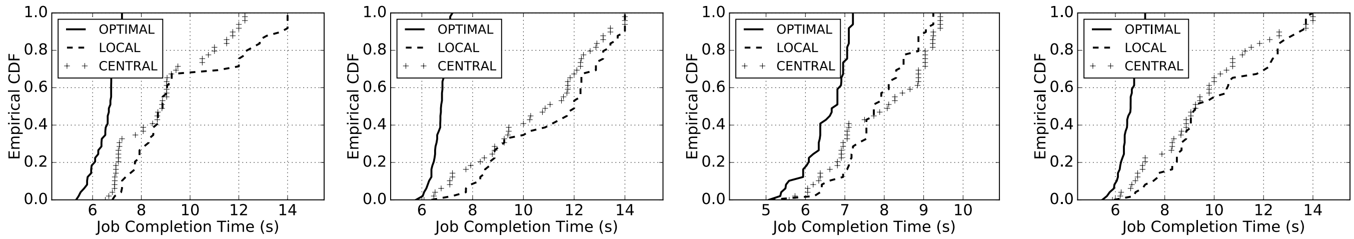
6.2 Large-Scale Simulations

We further present the results and analysis of our extensive simulations on our scheduling algorithm, to evaluate its effectiveness and performance at a finer granularity.

To allow simulations at a large scale, we implemented our simulator in `JAVA`, using the efficient `CPLEX` optimizer [13] to solve our LP problems. We conducted an extensive set of simulations, each with 50 and 100 concurrent jobs, respectively. The setting of the simulated inter-datacenter network is in consistent with our real-world experiment setting, where 6 geographically distributed datacenters are interconnected by links with bandwidth capacities shown in Table 1. For simplicity, each job has 10 tasks and each task has three partitions of input data, which are randomly distributed across all the datacenters. The size of input data is uniformly distributed in the range of $[50, 600]$ MB. The total resource capacity, *i.e.*, the total amount of available computing slots, across all the datacenters is set as $1, 1.1, 1.5, 2.5, 5$ and 10 times the total number of tasks (represented as $1X$, $1.1X$, $1.5X$, $2.5X$, $5X$ and $10X$), respectively in each group of simulation, to represent the different degrees of resource competition.

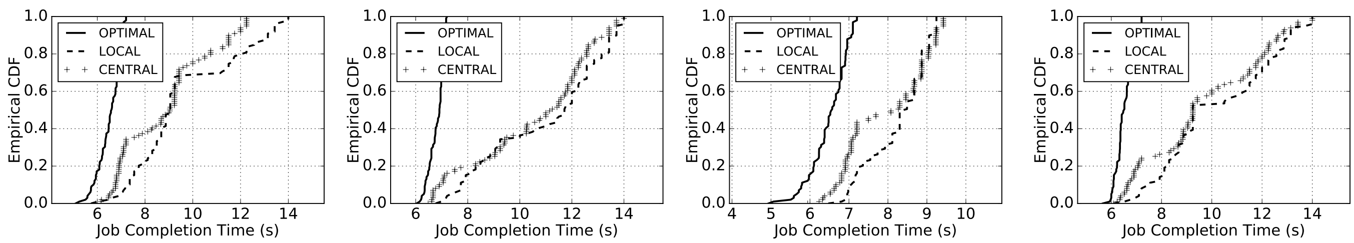
We compare our proposed algorithm, referred to as *OPTIMAL*, with two baselines — *LOCAL* and *CENTRAL*. *LOCAL* refers to the default Spark scheduling algorithm applied in the wide-area scenario, which tries to allocate tasks to the datacenter where their input data are stored. If such a data locality can not be satisfied for the lack of sufficient resources, *LOCAL* will randomly select a datacenter with available resources for task assignment. *CENTRAL* refers to the algorithm which attempts to assign all the tasks of a job to one or a few main

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering



(a) Even and Loose ($2X$) (b) Even and Tight ($1.1X$) (c) Skew and Loose ($2X$) (d) Skew and Tight ($1.1X$)

Fig. 10: The CDFs of job completion times for 50 concurrent jobs, achieved with three algorithms, with different resource availability.



(a) Even and Loose ($2X$) (b) Even and Tight ($1.1X$) (c) Skew and Loose ($2X$) (d) Skew and Tight ($1.1X$)

Fig. 11: The CDFs of job completion times for 100 concurrent jobs, achieved with three algorithms, with different resource availability.

datacenters as long as computing slots are available. It represents the most naive way of running data analytics in the wide area, which aggregates all the input data to a central datacenter and thus run all the tasks within a single cluster.

Our primary evaluation metrics are the completion times of concurrent jobs and the improvement ratio of the worst performance among concurrent jobs. To be more specific, for a job whose completion time is *opt* with our algorithm and *baseline* with a baseline algorithm, the improvement ratio is calculated as $(opt - baseline)/baseline \times 100\%$, of which the maximum is 100%.

Impact of Resource Availability. We first present a group of results with different settings of resource availability. Fig. 10 and Fig. 11 illustrate the empirical CDFs of job completion times across all the 50 and 100 concurrent jobs, respectively. Each subfigure presents the CDFs of job completion times achieved with the three comparing algorithms (*OPTIMAL*, *LOCAL* and *CENTRAL*), given different settings of resource distribution and competition degree. It is easily observed in all the figures that our algorithm always outperforms the baselines with respect to improving the worst job completion time.

Competition Degree: Tight vs. Loose. Fig. 10a and Fig. 10b illustrate the completion times of 50 concurrent jobs when all the available computing slots are evenly distributed across all the datacenters. The degrees of resource competition are set as $2X$ (*loose*) and $1.1X$ (*tight*) for Fig. 10a and Fig. 10b, representing that the total numbers of available slots are 2 and 1.1 times the total number of tasks, respectively. As we observe, when the resource competition becomes more intense, *i.e.*, there is a tight budget of resource, the difference between our algorithm and the baselines becomes more obvious. This implies that our algorithm shows more benefits in resolving competitions with our max-min fairness when the budget of resource is tight. In a similar vein, when the available computing slots are skew in distribution, our *OPTIMAL* also achieves shorter job completion times than the baselines, and the advantage has an increasing trend with a tighter budget of resources, as demonstrated by Fig. 10c and Fig. 10d. The similar analysis applies to Fig. 11 when the number of concurrent jobs is increased to 100.

Resource Distribution: Skew vs. Even. Given a fixed degree of resource competition, our algorithm shows more advantage in the case of a even resource distribution, compared with a skew one. This can be easily demonstrated by comparing Fig. 10a and Fig. 10c for the *loose* case, and comparing Fig. 10b and Fig. 10d for the *tight* case. Also, Fig. 11 shows a similar pattern. The explanation for such a pattern is that when the resource is evenly distributed, our algorithm has a larger space for optimizing the task placement, and thus exhibit more performance improvement.

Average Improvement Ratio. Apart from the detailed case study for Fig. 10 and Fig. 11, we further present performance improvement ratios achieved with 6 degrees — $1X$, $1.1X$, $1.5X$, $2.5X$, $5X$ and $10X$ — of resource competition, for 50 and 100 jobs, respectively, where the available computing slots are randomly distributed across the datacenters. Specifically, the performance improvement ratio is the percentage of reduction in the worst completion time among all the concurrent jobs achieved by our *OPTIMAL*, compared with the baseline *LOCAL*. Fig. 12 and Fig. 13 show the empirical CDFs of the improvement ratio over 20 runs, for 50 and 100 concurrent jobs, respectively. Moreover, the average

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSE.2018.2795580, IEEE Transactions on Network Science and Engineering

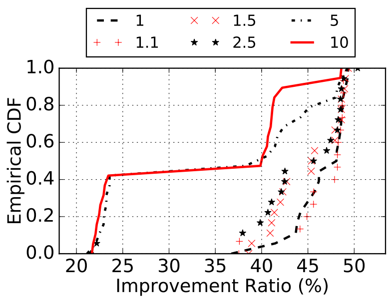


Fig. 12: CDFs of improvement ratios for 50 jobs over 20 runs. (*OPTIMAL* vs. *LOCAL*)

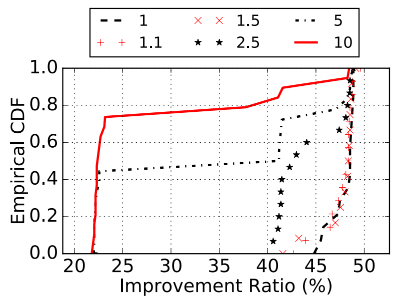


Fig. 13: CDFs of improvement ratios for 100 jobs over 20 runs. (*OPTIMAL* vs. *LOCAL*)

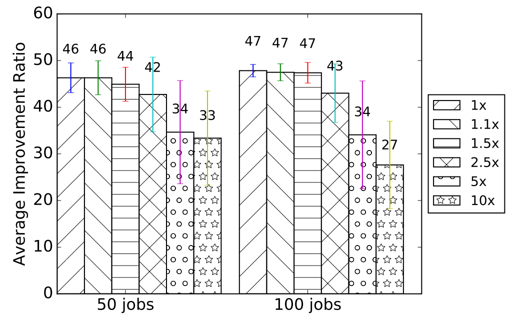
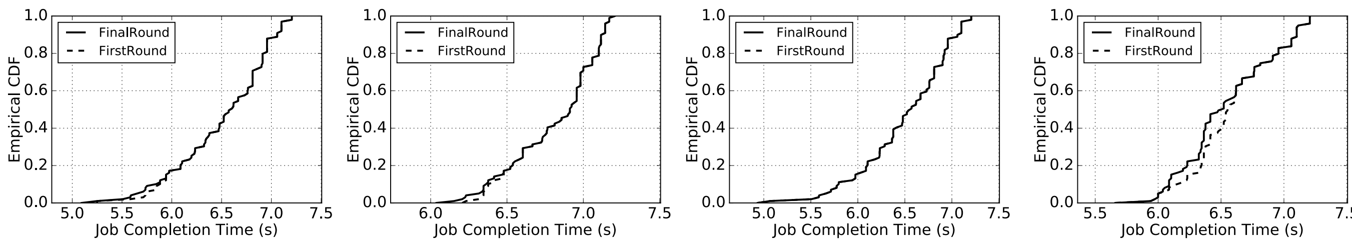


Fig. 14: Average improvement ratios with different resource availability for 50 and 100 jobs.



(a) Even and Loose ($2X$) (b) Even and Tight ($1.1X$) (c) Skew and Loose ($2X$) (d) Skew and Tight ($1.1X$)

Fig. 15: The CDFs of job completion times for 100 concurrent jobs, achieved in the first round and the final round of *OPTIMAL*.

performance improvement ratios are presented in Fig. 14 for intuitive comparison.

Clearly, our algorithm achieves a performance improvement of 27% to 47% over *Local*. With a larger degree of resource competition (*i.e.*, with more available computing slots), the improvement ratio becomes smaller. The reason is that our algorithm always seeks for the optimal solution in minimizing the worst job completion time, despite the degree of competition. However, the performance of *LOCAL* is impacted by the resource availability. With more resources, *LOCAL* can achieve data locality for more tasks, which avoids stragglers and improves job completion times. Therefore, the advantage of our algorithm becomes less obvious for the $2.5X$, $5X$ and $10X$ cases. The higher standard deviation for these cases can be explained by the randomness in placing tasks that can not achieve data locality in *LOCAL*. Though the total amount of slots is larger, some tasks may still fail to achieve data locality, as the resources are skew in distribution.

Improvement Over Rounds with *OPTIMAL*. Now we focus on the behavior of our algorithm by investigating the calculated task scheduling decisions over rounds. To be particular, we calculate the job completion times if scheduled with the temporary decision calculated in the first round and the final decision after the final round, respectively. We find that the job completion times achieved in these two rounds are the same for most cases. For the remaining cases, the improvement achieved in the final round over the first round is marginal. We present the CDFs of completion times for 100 concurrent jobs achieved in the first round and the final round of our algorithm, respectively, in Fig. 15. The settings of the four subfigures are the same with Fig. 11. As clearly illustrated, the performance in the final round shows no difference with the first found in Fig. 15c, and is slightly better in Fig. 15a, 15b and 15d. The improvement is due to the iterative optimization of our algorithm in improving performance of some jobs without impacting others, which has been clearly illustrated in Sec. 5. These figures imply that the optimization in the first round is the major contributor to the performance improvement of our algorithm. The space for further improvement over later rounds depends on specific cases and is not large.

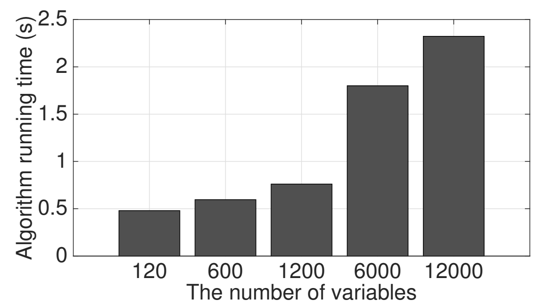


Fig. 16: The computation times of our algorithm in large-scale simulations, with different scales.

Scalability. Furthermore, as in our real-world experiment, we have measured the running times with different problem scales in our simulations to evaluate the scalability of our algorithm. The measured computation time with each problem scale is averaged over

10 runs. Fig. 16 illustrates the computation times when the number of variables ranges from 120 to 12000. As the CPLEX solver used in simulations is more efficient, solving the problem with 120 variables only takes less than 0.5 seconds, which is much faster than the solver in Breeze used in our experiment. When the number of variables grows beyond ten thousands, the computation time is still less than 2.5 seconds.

7 Discussion

Adaptation to Network Heterogeneity. Distributed data analytics depends heavily on the reliability and the performance of the underlying networks, due mainly to its abundant demand for data exchange among tasks. However, the networks, being best-effort and shared among many, can hardly deliver consistent or predictable Quality of Service, especially in the wide area networks. As a result, data analytics jobs usually suffer from performance degradation due to *spatial* and *temporal* network heterogeneity: different links can offer different capacities at the same time, while the available capacity of a link can vary over time. It remains an open problem to deal with these two types of heterogeneity in data analytics systems adaptively and in real-time. Solving this problem requires to identify the heterogeneity effectively and to adjust the job scheduling with the awareness of application-level workload. Our work offers a preliminary solution which accounts for the spatial network heterogeneity. To be adaptive to the temporal heterogeneity, we can monitor the bandwidth and run our algorithm periodically or when the bandwidth dynamics exceeds a certain threshold.

Economic-Based Resource Allocation. It would be interesting to investigate the problem of task assignment for data analytic jobs among geo-distributed datacenters from the game theoretic perspective. For example, in the framework of auctions, each job can be viewed as a buyer bidding for multiple goods, in the form of computing slots, from multiple datacenters. Different from conventional resource allocation where the utility of a user is determined by the summation of the total amount of resources received, allocating resources for data analytic jobs suffers from the challenge that the job performance is determined by the slowest task, which means that the utility is not determined by a simple summation of resources. Such a non-additive nature may significantly change the problem structure and introduce complexities in identifying the optimal solution. It is an interesting direction to be explored in our future work.

8 Related Work

With increasingly large volumes of data generated globally and stored in geo-distributed datacenters, it has received an increasing amount of research attention to deploying data analytic jobs across multiple datacenters. Based on their objectives, existing efforts can be roughly divided into two categories: reducing the amount of inter-datacenter network traffic to save operation costs, and reducing the job completion time to improve application performance.

Vulimiri *et al.* [3], [14] took the initiative to reduce the amount of data to be moved across datacenters when running geo-distributed data analytic jobs. To reduce the bandwidth cost, they formulated an integer programming problem to optimize the query execution plan and the data replication strategy. They also took advantage of the abundant storage resources to aggressively cache results of queries, to be leveraged by subsequent queries to reduce data transfers. Pixida [15] proposed to divide the DAG of a job into several parts, each to be executed in a datacenter, with the objective of minimizing the total amount of traffic among these divided parts. Despite reducing traffic across datacenters, these solutions do not necessarily shorten job completion times, as bandwidth availability varies across different links and over time.

As a representative work in the second category, Iridium [4] proposed an online heuristic to place both data and tasks across datacenters. Unfortunately, it assumes that the wide-area network that interconnects datacenters is free of congestion, which is far from realistic. Flutter [5] removed this unrealistic assumption, formulated a lexicographical minimization problem of task assignment for a single stage of one job, and obtained its optimal solution. However, all existing works focused on assigning tasks in a single job, without considering the inherent competition for resources among concurrent jobs. Despite using a similar theoretical foundation as [5], our problem considers multiple jobs, and is therefore remarkably different and more challenging.

Accounting for the scenario of multiple jobs sharing geo-distributed datacenters, Hung *et al.* [16] proposed a greedy scheduling heuristic to make job scheduling decisions across geo-distributed datacenters, with an objective of reducing the average job completion time. However, it assumes that the task assignment is predetermined, and the scheduling decision is the execution order of all the assigned tasks in each datacenter. Therefore, despite sharing a similar context of considering multiple jobs sharing the same pool of computing resources in geo-distributed datacenters, this work is orthogonal to our work, which aims to determine the best possible placement for tasks of all the sharing jobs with the consideration of fairness. As an extension to our previous conference paper [17], we have added results and analysis of an extensive set of large-scale simulations, to comprehensively explore the behavior of our algorithm.

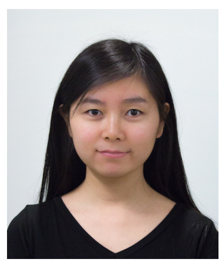
There are plenty of existing efforts [18]–[20] related to task assignment and job scheduling in big data analytic frameworks. To reduce job completion times, they proposed to improve data locality and fairness [18], [19], and to mitigate the negative impact of tasks that progress slowly, called stragglers ([20]). However, they are all designed for frameworks deployed in a single datacenter, and do not work effectively across multiple datacenters.

9 Concluding Remarks

In this paper, we have conducted a theoretical study of the task assignment problem among competing data analytic jobs, whose input data are distributed across geo-distributed datacenters. With tasks from multiple jobs competing for the computing slots in each datacenter, we have designed and implemented a new optimal scheduler to assign tasks across these datacenters, in order to better satisfy job requirements with max-min fairness achieved across their job completion times. To achieve this objective, we first formulated a lexicographical minimization problem to optimize all the job completion times, which is challenging due to the inherent complexity of both multi-objective and discrete optimizations. To address these challenges, we started from the single-objective subproblem and transformed it into an equivalent linear programming (LP) problem to be efficiently solved in practice, based on an in-depth investigation of the problem structure. An algorithm is further designed to repeatedly solve an updated version of the LP subproblems, which would eventually optimize all the job performance with max-min fairness achieved. Last but not the least, we have implemented our performance-optimal scheduler in the popular Spark framework, and demonstrated convincing evidence on the effectiveness of our new algorithm using both real-world experiments and large-scale simulations.

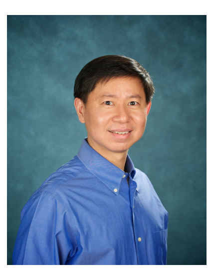
References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.
[3] A. Vulimiri, C. Curino, P. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global Analytics in the Face of Bandwidth and Regulatory Constraints," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.
[4] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low Latency Geo-Distributed Data Analytics," in *Proc. ACM SIGCOMM*, 2015.
[5] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling Tasks Closer to Data Across Geo-Distributed Datacenters," in *Proc. IEEE INFOCOM*, 2016.
[6] B. Korte and J. Vygen, *Combinatorial Optimization: Theory and Algorithms*, 3rd ed., ser. Algorithms and Combinatorics. Springer, 2006, vol. 21, ch. 5, p. 104.
[7] R. Meyer, "A Class of Nonlinear Integer Programs Solvable by A Single Linear Program," *SIAM Journal on Control and Optimization*, vol. 15, no. 6, pp. 935–946, 1977.
[8] E. Andersen and K. Andersen, "The MOSEK Interior Point Optimizer for Linear Programming: an Implementation of the Homogeneous Algorithm," in *High Performance Optimization*. Springer, 2000, pp. 197–232.
[9] Y. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.
[10] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the Best VM across Multiple Public Clouds: A Data-Driven Performance Modeling Approach," in *Proc. ACM Symposium on Cloud Computing (SOCC)*, 2017.
[11] Breeze: A Numerical Processing Library for Scala. [Online]. Available: http://www.scalanlp.org
[12] Hadoop. [Online]. Available: https://hadoop.apache.org/
[13] CPLEX Optimizer: High-Performance Mathematical Programming Solver. [Online]. Available: https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
[14] A. Vulimiri, C. Curino, P. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for A Geo-Distributed Data-Intensive World," in *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2015.
[15] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics," *VLDB Endowment*, vol. 9, no. 2, pp. 72–83, 2015.
[16] C. Hung, L. Golubchik, and M. Yu, "Scheduling Jobs Across Geo-Distributed Datacenters," in *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2015.
[17] L. Chen, S. Liu, B. Li, and B. Li, "Scheduling Jobs across Geo-Distributed Datacenters with Max-Min Fairness," in *Proc. IEEE INFOCOM*, 2017.
[18] 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 *Proc. ACM European Conference on Computer Systems*, 2010, pp. 265–278.
[19] B. Hindman, A. Konwinski, M. Zaharia, and et al., "Mesos: A Platform for Fine-Grained Resource Sharing in The Data Center," in *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.
[20] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale," in *Proc. ACM SIGCOMM*, 2015.



Li Chen is currently pursuing her Ph.D. degree at the Department of Electrical and Computer Engineering, University of Toronto, where she received her M.A.Sc. degree in 2014. She received her B.Engr. degree from the Department of Computer Science and Technology, Huazhong University of Science and Technology, China, in 2012. Her research interests include big data analytics systems, cloud computing, datacenter networking and resource allocation.

Shuhao Liu is currently a Ph.D. student in the Department of Electrical and Computer Engineering, University of Toronto. He received his B.Eng. degree from Tsinghua University in 2012. His current research interests include software-defined networking and big data analytics.



Baochun Li received his Ph.D. degree from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 2000. Since then, he has been with the Department of Electrical and Computer Engineering at the University of Toronto, where he is currently a Professor. He holds the Bell Canada Endowed Chair in Computer Engineering since August 2005. His research interests include large-scale distributed systems, cloud computing, peer-to-peer networks, applications of network coding, and wireless networks. He is a member of the ACM and a Fellow of the IEEE.

Bo Li is a Professor in the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He holds the Cheung Kong Chair Professor in Shanghai Jiao Tong University. Prior to that, he was with IBM Networking System Division, Research Triangle Park, North Carolina. He was an adjunct researcher at Microsoft Research Asia-MSRA and was a visiting scientist at Microsoft Advanced Technology Center (ATC). He has been a technical advisor for ChinaCache Corp. (NASDAQ CCIH) since 2007. He is an adjunct professor in Huazhong University of Science and Technology, Wuhan, China. His recent research interests include: large-scale content distribution in the Internet, Peer-to-Peer media streaming, the Internet topology, cloud computing, green computing and communications. He is a Fellow of IEEE for "contribution to content distributions via the Internet". He received the Young Investigator Award from the National Natural Science Foundation of China (NSFC) in 2004. He served as a Distinguished Lecturer for IEEE Communications Society (2006-2007). He was a co-recipient for three Best Paper Awards from IEEE, and the Best System Track Paper in ACM Multimedia (2009).