Time- and Cost- Efficient Task Scheduling Across Geo-Distributed Data Centers

Zhiming Hu, *Member, IEEE,* Baochun Li, *Fellow, IEEE,* and Jun Luo, *Member, IEEE*

Abstract—Typically called big data processing, analyzing large volumes of data from geographically distributed regions with machine learning algorithms has emerged as an important analytical tool for governments and multinational corporations. The traditional wisdom calls for the collection of all the data across the world to a central data center location, to be processed using data-parallel applications. This is neither efficient nor practical as the volume of data grows exponentially. Rather than transferring data, we believe that computation tasks should be scheduled near the data, while data should be processed with a minimum amount of transfers across data centers. In this paper, we design and implement *Flutter*, a new task scheduling algorithm that reduces both the completion times and the network costs of big data processing jobs across geographically distributed data centers. To cater to the specific characteristics of data-parallel applications, in the case of optimizing the job completion times only, we first formulate our problem as a lexicographical min-max integer linear programming (ILP) problem, and then transform the ILP problem into a nonlinear program problem with a separable convex objective function and a totally unimodular constraint matrix, which can be further solved using a standard linear programming solver efficiently in an online fashion. In the case of improving both time- and cost- efficiency, we formulate the general problem as an ILP problem and we find out that solving an LP problem can achieve the same goal in the real practice. Our implementation of *Flutter* is based on Apache Spark, a modern framework popular for big data processing. Our experimental results have shown convincing evidence that *Flutter* can shorten both job completion times and network costs by a substantial margin.

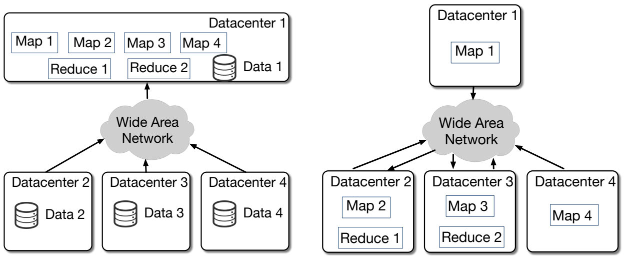
Index Terms—big data processing, task scheduling, cloud computing

1 Introduction

It has now become commonly accepted that the volume of data — from end users, sensors, and algorithms alike — has been growing exponentially, and data is mostly stored in geographically distributed data centers around the world. *Big data processing* refers to applications that apply machine learning algorithms to process such large volumes of data, typically supported by modern data-parallel frameworks such as Spark. Needless to say, big data processing has become routine in governments and multinational corporations, especially those in the business of social media and Internet advertising.

Big data processing over geo-distributed data attracts much attention recently, and it can provide several benefits. For example, network operators must analyze traffic in multiple data centers to detect probes from attackers disguised as normal traffic [2]. Such analysis must extract global patterns (e.g., frequent IPs) from logs at each site. Nokia reports that site-local analysis would miss many devastating attacks [3]. Other use cases for geo-distributed processing include personal recommendations based on the geo-distributed user activity logs.

To process large volumes of data that are geographically distributed, we will traditionally need to transfer all the data to be processed to a single data center, so that they can be processed in a centralized fashion [4]. However, at times, such traditional wisdom may not be practically feasible. First, it may not be practical to move user data across country boundaries, due to legal reasons or privacy concerns [5]. Second, the cost, regarding both bandwidth cost and time, to move large volumes of data across geo-distributed data centers may become prohibitive as the amount of data grows exponentially.



(a) Traditional approach. (b) Our approach.

Fig. 1: Processing data locally by moving computation tasks: an illustrating example.

It has been pointed out that [4], [5], [6], rather than transferring data across data centers, it may be a better design to move computation tasks to where the data is so that data can be processed locally within the same data center. Of course, the intermediate results after such processing may still need to be transferred across data centers, but they are typically much smaller in size, significantly reducing the cost of data transfers. An example showing the benefits of processing big data over geo-distributed data centers is shown in Fig. 1.

- *Zhiming Hu and Baochun Li are with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, M5S 2E4.*
  E-mail: zhiming@ece.utoronto.ca, bli@ece.toronto.edu.
- *Jun Luo is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, 639798.*
  E-mail: junluo@ntu.edu.sg.
- *Preliminary results were presented in Proceedings of the IEEE INFOCOM, 2016 [1].*

The fundamental objective, in general, is to minimize the job completion times in big data processing applications, by placing the tasks at their respective best possible data centers. However, previous works (*e.g.,* [4]) were designed with assumptions that were often unrealistic — such as bottlenecks do not occur on inter-data center links.

Intuitively, it may be a step towards the right direction to design an offline optimal task scheduling algorithm, so that the job completion times are globally minimized. However, such offline optimization inevitably relies upon *a priori* knowledge of task execution times and transfer times of intermediate results, neither of which is readily available without sophisticated prediction algorithms. Even if such knowledge is available, a large data processing job in Spark may involve a directed acyclic graph (DAG) with hundreds of tasks; and optimal solutions for scheduling such a DAG is NP-Complete in general [7].

Besides job completion times, the cost of data transfers among different data centers is a big concern for some workloads as the cost could be very high [5]. More specifically, for workloads without network budget constraints, job completion time is the sole goal, and we can optimize it directly. While for workloads with network budget constraints, achieving the optimal job completion time does not necessarily result in the optimal cost of data transfers. This is because the job completion time only cares about the bottleneck links, while cost savings of data transfers can only be accomplished by reducing the total data transfers on all the links across data centers. Thus, in this case, we aim to optimize the job completion time at a reasonable bandwidth cost and design a tunable network budget for those workloads.

In this paper, we develop and implement *Flutter,* a new system to schedule tasks across data centers over the wide area for both time- and cost- efficiency. Our primary focus when designing *Flutter* is on practicality and real-world implementation, rather than on the optimality of our results. To be practical, *Flutter* is first and foremost designed as an *online* scheduling algorithm, making adjustments on-the-fly based on the current job progress. *Flutter* is also intended to be *stage-aware*: it minimizes the completion time of each stage in a job, which corresponds to the slowest of the completion time of the constituent tasks in the stage.

Practicality also implies that our algorithms in *Flutter* would need to be efficient at runtime. Our problems of stage-aware online scheduling can be formulated as lexicographical min-max integer linear programming (ILP) problems. A highlight of this paper is that, for the case of without network budget constraints, after transforming the problem into a nonlinear program, we show that it has a separable convex objective function and a totally unimodular constraint matrix, which can then be solved using a standard linear programming solver efficiently, and in an online fashion. For the other case of considering the network budget constraints, we also show that it can produce near optimal scheduling results by solving a linear programming (LP) problem.

To demonstrate that it is amenable to practical implementations, we have implemented *Flutter* based on Apache Spark, a modern framework for big data processing. Our experimental results on a production wide-area network with geo-distributed servers have shown convincing evidence that *Flutter* can shorten job completion times and reduce network costs by a substantial margin.

2 Flutter: Motivation and Problem Formulation

In this section, we first show the backgrounds and motivations of the proposed algorithms. We then illustrate our mathematic models for the problems. Finally, we formally formulate the problems for workloads with/without network budget constraints, respectively.

2.1 Background and Motivation

Here we show some characteristics of inter-data center networks regarding the bandwidths and pricing for data transfers, which serve as our motivations.

2.1.1 Bandwidths Across Data Centers

To motivate our work, we begin with a real-world experiment, with *Virtual Machines* (VMs) initiated and distributed in four representative regions in Amazon EC2: EU (Frankfurt), US East (N. Virginia), US West (Oregon), and Asia Pacific (Singapore). All the VM instances we used are `m3.xlarge`, with four cores and 15 GB of main memory each. To illustrate the available capacities on inter-data center links, we have measured the bandwidths available between VMs across data centers using the `iperf` utility and our results are shown in Table 1.

TABLE 1: Available bandwidths between VMs across geographically distributed data centers as of July 2015.

	EU	US-East	US-West	Singapore
EU	946 Mbps	136 Mpbs	76.3 Mbps	**49.3 Mbps**
US-East	-	1.01 Gbps	**175 Mbps**	52.6 Mbps
US-West	-	-	945 Mbps	76.9 Mbps
Singapore	-	-	-	945 Mbps

From this table, we can make two observations with convincing evidence. On the one hand, when VMs in the same data center communicate with each other across the intra-data center network, the available bandwidth is consistently high, at around 1 Gbps, which is sufficient for typical Spark-based data-parallel applications [8]. On the other hand, bandwidths between VMs across data centers are an order of magnitude lower and vary significantly for different inter-data center links between VMs. For example, the highest bandwidth in the table is 175 Mbps, while the lowest is only 49 Mbps.

Our observations have clearly implied that transfer times of intermediate results across data centers can quickly become the bottleneck when it comes to job completion times if we run the same data-parallel application across different data centers. Scheduling tasks carefully to the best possible data centers is, therefore, important to utilize available inter-data center bandwidth better; and more so when the inter-data center bandwidths are lower and more divergent. *Flutter* is first and foremost designed to be *network-aware*, in that tasks can be scheduled across geo-distributed data centers with the awareness of available inter-data center bandwidth.

2.1.2 Bandwidth Costs Across Data Centers

Besides the differences of bandwidths across data centers, the bandwidth pricing is also an important factor for workloads with network budget constraints. Let us first look at the pricing of data transfers among different data centers. We show the pricing for data transfers of Microsoft Azure [9] and Amazon Web Service (AWS) [10] in Table 2 and Table 3, respectively. The units for the pricing are both US dollars/Gigabytes. We should also note that the pricing is only for outbound traffic to other data centers. Inbound traffic is free for almost all the public cloud providers like AWS, Azure and Google Compute Engine [11].

TABLE 2: Bandwidth costs across geographically distributed data centers in Azure [12].

US and EU	Asia Pacific, Japan and Australia	Brazil
0.087	0.138	0.181

TABLE 3: Bandwidth costs across geographically distributed data centers in amazon web service (AWS) [13].

US and EU	Singapore and Tokyo	Sydney	San Paulo
0.02	0.09	0.14	0.16

Based on the tables, we can clearly see that the network pricing would vary a lot in both of the cloud platforms. More specifically, in Azure, the most expensive pricing of data transfers among data centers per GB is more than **two times** to the cheapest pricing. Moreover, the most expensive pricing is **eight times** to the cheapest one in AWS. Both of the cases strongly imply that we should also avoid scheduling the tasks to data centers that would incur high costs of data transfers. In other words, a wise scheduling decision should also consider the pricing differences and the pricing policies of data transfers among data centers besides the differences of bandwidths among data centers if there are network budget constraints.

To illustrate that the best scheduling choice for task/job completion time would not necessarily result in the lowest cost of data transfers, we provide a simple motivating example in Fig. 2. In this example, we can see that if we only consider the task completion time, it would take the same amount of time to get all its inputs. Then it would achieve the same task completion time because no matter where we schedule the task, we need to transfer the other input from the other data center and the size of inputs are the same. However, we can easily see that transferring the input from data center 1 to data center 2 is much more expensive than the other way around. Therefore, scheduling the task to data center 1 is apparently a better solution if we also take the cost for data transfers among data centers into consideration. This motivation example strongly implies the need of considering the network budget and pricing policies in task scheduling especially for workloads with network budget constraints.

2.2 The Model

To formulate the problem that we wish to solve with the design of *Flutter*, we revisit the current task scheduling disciplines in existing data-parallel frameworks that support big data processing, taking Spark [14] as an example. In Spark, a job can be represented by a Directed Acyclic Graph (DAG) $G = (\mathcal{V}, \mathcal{E})$. Each node $v \in \mathcal{V}$ represents a task; each directed edge $e \in \mathcal{E}$ indicates a precedence constraint, and the weight of e represents the transfer time of intermediate results from the source node to the destination node of e.

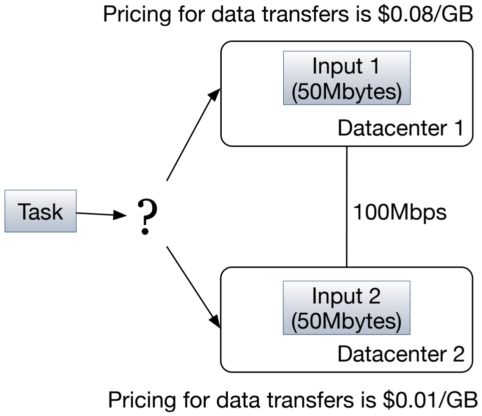

Pricing for data transfers is $0.08/GB
Input 1 (50Mbytes)
Datacenter 1
Task
?
100Mbps
Input 2 (50Mbytes)
Datacenter 2
Pricing for data transfers is $0.01/GB

Fig. 2: A motivating example. Schedule the tasks when we should both consider the bandwidth and costs for transferring data. The task needs two inputs: input 1 and input 2.

Scheduling all the tasks in the DAG to some worker nodes — while minimizing the completion time of the job — is NP-Complete in general [7], and is neither efficient nor practical. Rather than scheduling all the tasks together, Spark schedules ready tasks stage by stage in an online fashion. As it is a much more convenient way of designing a task scheduler, *Flutter* follows suit and only schedules the tasks within the same stage to geo-distributed data centers, rather than considering all the ready tasks in the DAG[1]. Here we denote the set of the index of tasks in a stage by $\mathcal{N} = \{1 \dots n\}$ and the set of the index of data centers by $\mathcal{D} = \{1 \dots d\}$. Therefore, the constraints in the problem can be listed as below.

Task placement constraints: x_{ij} denotes whether i-th task will be scheduled to the j-th data center. $x_{ij} = 1$ indicates the assignment of the i-th task to j-th data center; otherwise $x_{ij} = 0$.

$$\sum_{j=1}^{d} x_{ij} = 1, \; \forall i \in \mathcal{N}, \tag{1}$$

$$x_{ij} \in \{0, 1\}, \; \forall i \in \mathcal{N}, \; \forall j \in \mathcal{D}. \tag{2}$$

The first constraint in Eq. (1) implies that each task should be scheduled to only one data center. The second constraint shows that x_{ij} is binary.

Capacity constraints: The number of tasks assigned to the j-th data center should not exceed the maximum number of tasks f_j that can be scheduled on the existing VMs in that data center. This constraint is shown in Eq. (3). Though it is indeed conceivable to launch new VMs on-demand, it takes around 100 seconds for most of the types of instances [15], in reality, to initiate and start a new VM, making it far from practical. The total number of tasks that

[1] This approach may result in sub-optimal performance regarding the job completion time of the whole job.

can be scheduled in one data center depends on the total number of CPU cores available in the data center and the number of cores needed for each task in Spark.

$$\sum_{i=1}^{n} x_{ij} \leq f_j, \ \forall j \in \mathcal{D}. \tag{3}$$

Transfer time constraints: We compute the transfer time of intermediate data if the i-th task is scheduled on the j-th data center in Eq. (4). c_{ij} is the transfer time to receive all the intermediate results. s_i and d_k represent the number of inputs for the i-th task and the index of the data center that has the k-th input, respectively. For example, let $m_{d_k j}$ denote the amount of data that need to be transferred from the d_k-th data center to the j-th data center if the i-th task is scheduled to the j-th data center. If $d_k = j$, then $m_{d_k j} = 0$. We let b_{uv} to denote the bandwidth between the u-th data center and the v-th data center, and assume that the network bandwidth $B_{d \times d} = \{b_{uv} | \ u, \ v = 1 \ldots d\}$ across all the data centers can be measured and is stable for a few minutes [4]. We can then compute the maximum transfer times for each possible way of scheduling the i-th task.

$$c_{ij} = \max_{k \in s_i} (m_{d_k j} / b_{d_k j}), \ \forall i \in \mathcal{N}, \ \forall j \in \mathcal{D}. \tag{4}$$

Bandwidth cost constraints: Eq. (5) is the constraint to limit the bandwidth cost for the workloads that have budget constraints. $bcost_{ij}$ denotes the bandwidth cost incurred by scheduling the i-th task to the j-th data center, and it can be calculated by the summation of the costs of obtaining all the intermediate inputs where $p_{d_k j}$ is the pricing for data transfers from d_k-th data center to j-th data center as shown in (6). *opt* is the optimal bandwidth cost in the scheme optimized for bandwidth cost. θ ($\theta \geq 1$) is designed to tune the budget. If $\theta = 1.0$, it implies a strict bandwidth budget, and $\theta > 1$, otherwise. As shown in Eq. (5), we propose to set the cost limits based on the optimal bandwidth cost and a tunable variable θ.

$$\sum_{i=1}^{n} \sum_{j=1}^{d} (x_{ij} \times bcost_{ij}) \leq \theta \cdot opt, \tag{5}$$

$$bcost_{ij} = \sum_{k \in s_i} m_{d_k j} \cdot p_{d_k j}, \ \forall i \in \mathcal{N}, \ \forall j \in \mathcal{D}. \tag{6}$$

The problem to obtain the optimal bandwidth cost *opt* is formulated as follows:

$$\begin{aligned} \text{minimize} \quad & opt = \sum_{i=1}^{n} \sum_{j=1}^{d} (x_{ij} \cdot bcost_{ij}) \\ \text{s.t.} \quad & (1), (2), (3), (6). \end{aligned} \tag{7}$$

This formulation shows a very straightforward LP problem, while it can help calculate the optimal bandwidth cost for each stage. Each time, we will first compute the optimal bandwidth cost for each stage, and then feed it to the constraints in Eq. (5).

2.3 Problem Formulation

Given the model, we know that we schedule the tasks and optimize the performance stage by stage. There is, however, one more complication when tasks within the same stage are to be scheduled. The complexity comes from the fact that the completion time of a stage in data-parallel jobs is determined by the completion time of the *slowest* task in that stage. Without awareness of the stage that a task belongs to, it may be scheduled to a data center with a much longer transfer time to receive all the intermediate results needed (due to capacity limitations on inter-data center links), slowing down not only the stage it belongs to, but the entire job as well.

More formally, *Flutter* should be designed to solve a *network-aware* and *stage-aware* online reduce task scheduling problem, formulated as a lexicographical min-max integer linear programming (ILP) problem **P1** as follows:

$$\begin{aligned} \underset{X}{\text{lexmin}} \quad & \max_{i,j} (x_{ij} \cdot (c_{ij} + e_{ij})) \\ \text{s.t.} \quad & (1), (2), (3), (4). \end{aligned} \tag{8}$$

where c_{ij} is the transfer time to receive all the intermediate results for task i, computed in Eq. (4). e_{ij} denotes the execution time of the i-th task in the j-th data center. Our objective is to minimize the maximum task completion time within a stage, including both the network transfer time and the task execution time. In other words, we optimize the *stage completion times*.

By far, we have formulated the problem for workloads without network budget constraints. However, there are some other workloads that only aim to optimize the performance given certain network budget constraints. To this end, we also formulate the problem for workloads with network budget constraints as **P2**:

$$\begin{aligned} \underset{X}{\text{lexmin}} \quad & \max_{i,j} (x_{ij} \cdot (c_{ij} + e_{ij})) \\ \text{s.t.} \quad & (1), (2), (3), (4), (5), (6). \end{aligned} \tag{9}$$

3 Task Scheduling across Geo-Distributed Data Centers

Given the problem formulations of our task scheduling problems across geo-distributed data centers for workloads with/without network budget constraints, we now study how to solve the problems efficiently, which is the key for the practicality of *Flutter* in the real data processing systems.

3.1 Workloads without Network Budget Constraints

In this section, we show how we solve the problem **P1** for workloads without network budget constraints. More specifically, we first propose to transform the lexicographical min-max integer problem in the formulation into a particular class of nonlinear programming problem. We then further change this special class of nonlinear programming problem into a *linear programming problem* (LP) that can be solved efficiently with standard LP solvers.

3.1.1 Transform into a Nonlinear Programming Problem

The special class of nonlinear programs that can be converted into an LP has two characteristics [16], [17], a separable convex objective function and a totally unimodular constraint matrix. We will show how we change our original formulation to meet these two conditions.

3.1.1.1 Separable Convex Objective Function: A function is separable convex if it can be represented as a summation of multiple convex functions with a single variable. To make this transformation, we first define the lexicographical order. Let $\mathbf{p}$ and $\mathbf{q}$ represent two integer vectors of length k. We define $\overrightarrow{\mathbf{p}}$ and $\overrightarrow{\mathbf{q}}$ as the sorted $\mathbf{p}$ and $\mathbf{q}$ with non-increasing order, respectively. If $\mathbf{p}$ is lexicographically less than $\mathbf{q}$, represented by $\mathbf{p} \prec \mathbf{q}$, it means that the first non-zero item of $\overrightarrow{\mathbf{p}} - \overrightarrow{\mathbf{q}}$ is negative. Then if $\mathbf{p}$ is lexicographically no greater than $\mathbf{q}$, denoted as $\mathbf{p} \preceq \mathbf{q}$, it is equivalent to $\mathbf{p} \prec \mathbf{q}$ or $\overrightarrow{\mathbf{p}} = \overrightarrow{\mathbf{q}}$.

Our objective function is to find a vector that is lexicographically minimal over all the vectors in the feasible region with their components rearranged in a non-increasing order. In our problem, if $\mathbf{p}$ is lexicographically less than $\mathbf{q}$, then vector $\mathbf{p}$ is a better solution for our lexicographical min-max problem. However, directly finding the lexicographically minimal vector is not an easy task, we discover that we can use a summation of exponents to preserve the lexicographical order among vectors. Consider the convex function $g : \mathbb{Z}^k \to R$ that has the form of

$$g(\boldsymbol{\alpha}) = \sum_{i=1}^{k} k^{\alpha_i},$$

where $\boldsymbol{\alpha} = \{\alpha_i \mid i = 1 \ldots k\}$ is an integer vector with length k. We prove that we can preserve the lexicographical order of vectors through $g : \mathbb{Z}^k \to R$ by the following lemma[2].

Lemma 1. For $\mathbf{p},\ \mathbf{q} \in \mathbb{Z}^k,\ \mathbf{p} \preceq \mathbf{q} \iff g(\mathbf{p}) \leq g(\mathbf{q})$.

Proof: We first prove that $\mathbf{p} \prec \mathbf{q} \implies g(\mathbf{p}) < g(\mathbf{q})$. We assume that the index of the first positive element of $\overrightarrow{\mathbf{q}} - \overrightarrow{\mathbf{p}}$ is r. As both vectors only have integral elements, $\overrightarrow{q}_r > \overrightarrow{p}_r$ implies $\overrightarrow{q}_r \geq \overrightarrow{p}_r + 1$. If $r = k$, this part is directly proved. Here we consider the case for $r \leq k - 1$. Then we have:

$$\begin{align}
g(\mathbf{q}) - g(\mathbf{p}) = \quad & g(\overrightarrow{\mathbf{q}}) - g(\overrightarrow{\mathbf{p}}) \tag{10}\\
= \quad & \sum_{i=1}^{k} k^{\overrightarrow{q}_i} - \sum_{i=1}^{k} k^{\overrightarrow{p}_i} \tag{11}\\
= \quad & \sum_{i=r}^{k} k^{\overrightarrow{q}_i} - \sum_{i=r}^{k} k^{\overrightarrow{p}_i} \tag{12}\\
\geq \quad & \sum_{i=r}^{k} k^{\overrightarrow{q}_i} - k \times k^{\overrightarrow{p}_r} \tag{13}\\
= \quad & (k^{\overrightarrow{q}_r} - k^{\overrightarrow{p}_r + 1}) + \sum_{i=r+1}^{k} k^{\overrightarrow{q}_i} \tag{14}\\
\geq \quad & \sum_{i=r+1}^{k} k^{\overrightarrow{q}_i} \tag{15}\\
> \quad & 0 \tag{16}
\end{align}$$

Hence the first part is proved.

We then show $g(\mathbf{p}) < g(\mathbf{q}) \implies \mathbf{p} \prec \mathbf{q}$ and we assume r is the index of first non-zero element in $\overrightarrow{\mathbf{q}} - \overrightarrow{\mathbf{p}}$, then $\overrightarrow{p}_i = \overrightarrow{q}_i$ for all $i < r$. Here we first consider the case when $r \geq 2$.

$$\begin{align}
g(\mathbf{q}) - g(\mathbf{p}) = \quad & g(\overrightarrow{\mathbf{q}}) - g(\overrightarrow{\mathbf{p}}) \tag{17}\\
= \quad & \sum_{i=1}^{k} k^{\overrightarrow{q}_i} - \sum_{i=1}^{k} k^{\overrightarrow{p}_i} \tag{18}\\
\leq \quad & \sum_{i=1}^{r-1} k^{\overrightarrow{q}_i} + (k + 1 - r) \times k^{\overrightarrow{q}_r} \tag{19}\\
& - \sum_{i=1}^{r-1} k^{\overrightarrow{p}_i} - k^{\overrightarrow{p}_r} \tag{20}\\
\leq \quad & (k + 1 - r) \times k^{\overrightarrow{q}_r} - k^{\overrightarrow{p}_r} \tag{21}
\end{align}$$

We can easily see that this inequation also holds when $r = 1$. Therefore if $g(\mathbf{q}) - g(\mathbf{p}) > 0$, then we have $(k + 1 - r) \times k^{\overrightarrow{q}_r} - k^{\overrightarrow{p}_r} > 0$. For $r = 1$, it implies $\overrightarrow{q}_r + 1 > \overrightarrow{p}_r$. If $\overrightarrow{q}_r < \overrightarrow{p}_r$, the previous inequation would not hold. $\overrightarrow{q}_r$ also does not equal $\overrightarrow{p}_r$ as r is the index of the first non-zero item in $\overrightarrow{\mathbf{q}} - \overrightarrow{\mathbf{p}}$. We then have $\overrightarrow{q}_r > \overrightarrow{p}_r$. For $r > 1$, $(k{+}1{-}r){\times}k^{\overrightarrow{q}_r}{-}k^{\overrightarrow{p}_r} > 0$ implies $\log_k(k{+}1{-}r){+}\overrightarrow{q}_r > \overrightarrow{p}_r$. Because $r > 1$, $\log_k(k + 1 - r)$ is less than 1 and $\overrightarrow{q}_r \neq \overrightarrow{p}_r$ because r is the index of first non-zero item in $\overrightarrow{\mathbf{q}} - \overrightarrow{\mathbf{p}}$. Thus we can also have $\overrightarrow{q}_r > \overrightarrow{p}_r$ when $r > 1$. In sum, $\overrightarrow{q}_r > \overrightarrow{p}_r$ for all $r \geq 1$. As a result, it can be concluded that $\mathbf{p} \prec \mathbf{q}$.

Regarding the equations, if $\overrightarrow{\mathbf{p}} = \overrightarrow{\mathbf{q}}$, it is straightforward to see that $g(\mathbf{q}) = g(\mathbf{p})$. Now if $g(\mathbf{q}) = g(\mathbf{p})$, let us prove whether we have $\overrightarrow{\mathbf{p}} = \overrightarrow{\mathbf{q}}$. Without loss of generality, we can assume that $\mathbf{p} \prec \mathbf{q}$ when $g(\mathbf{q}) = g(\mathbf{p})$. While if $\mathbf{p} \prec \mathbf{q}$, then we have $g(\mathbf{p}) < g(\mathbf{q})$ based on previous proofs, which contradicts to the assumption. Thus if $g(\mathbf{q}) = g(\mathbf{p})$, we also have $\overrightarrow{\mathbf{p}} = \overrightarrow{\mathbf{q}}$. □

Let $\mathbf{h}(X)$ denote the vector in the objective function of our problem in Eq. (8). Then our problem can be denoted by $\underset{X}{\text{lexmin}}\ (\max\ \mathbf{h}(X))$. Based on **Lemma 1**, the objective function of our problem can be further replaced by $\min\ g(\mathbf{h}(X))$, which is

$$\min\ \sum_{i}^{n} \sum_{j}^{d} k^{x_{ij} \cdot (c_{ij} + e_{ij})}, \tag{22}$$

where k equals nd, which is the length of vectors in the solution space of the problem in our formulation.

We can clearly see that each term of summation in Eq. (22) is an exponential function, which is convex. Therefore this new objective function consists of a separable convex objective function. Now let us see whether the coefficients in the constraints of our formulation can form a totally unimodular matrix.

3.1.1.2 Totally Unimodular Constraint Matrix: A totally unimodular matrix is an important concept as it can quickly determine whether an LP is integral, which means that the LP would only have integral optimum if it has any. For instance, if a problem has the form of $\{\min cx \mid AX \leq \mathbf{b},\ x > 0\}$, where A is a totally unimodular matrix and $\mathbf{b}$ is an integral vector, then the optimal solutions for this problem must be integral. The reason is that in this case, the feasible region $\{x \mid\ AX \leq \mathbf{b},\ x > 0\}$ is an integral polyhedron, which has only integral extreme points. Hence in our case, if we can prove that the coefficients in the constraints of our formulation form a totally

[2] Since scaling the coefficients of x_{ij} would not change the optimal solution, we can always make the coefficients to be integers.

unimodular matrix, then our problem would only have integral solutions. We prove that the constraint matrix in our problem formulation forms a totally unimodular matrix by the following lemma.

Lemma 2. The coefficients of the constraints (1) and (3) form a totally unimodular matrix.

Proof: A totally unimodular matrix is a $m \times r$ matrix $A = \{a_{ij} \mid i = 1 \dots m,\ j = 1 \dots r\}$ that meets the following two conditions. First, all of its elements must be selected from {-1, 0, 1}. It is straightforward to see that all the elements in the coefficients of our constraints are 0 or 1, so it meets the first condition. The second condition is that for any subset of rows $\mathcal{I} \in \{1 \dots m\}$, it can be separated into two sets $\mathcal{I}_1, \mathcal{I}_2$ such that $\| \sum_{i \in \mathcal{I}_1} a_{ij} - \sum_{i \in \mathcal{I}_2} a_{ij} \| \leq 1$. In our formulation, we can take the variable $X = \{x_{ij} \mid i = 1 \dots n,\ j = 1 \dots d\}$ as a $nd \times 1$ vector, then we can write down the constraint matrix in (1) and (3), respectively. We can then find out that for these two matrices, the sum over all the rows in each matrix both equal a $1 \times nd$ vector whose entries are all equal to 1. For any subset $\mathcal{I}$ of the matrix formed by the co-efficients in constraint (1) and (3), we can always assign the rows related to (1) to $\mathcal{I}_1$, and the rows related to (3) to $\mathcal{I}_2$. In this case, as both $\sum_{i \in \mathcal{I}_1} a_{ij}$ and $\sum_{i \in \mathcal{I}_2} a_{ij}$ are smaller than a $1 \times nd$ vector with nd 1s, we will always have $\| \sum_{i \in \mathcal{I}_1} a_{ij} - \sum_{i \in \mathcal{I}_2} a_{ij} \| \leq 1$. Then this lemma got proven. □

3.1.2 *Transform the Nonlinear Programming Problem into an LP*

We have transformed our integer programming problem into a nonlinear programming with a separable convex function. We have also shown that the coefficients in the constraints of our formulation form a totally unimodular matrix. Now we can further transform the nonlinear programming problem into an LP based on the method proposed in [16], [17]. In this transformation, the optimal solutions would not change. The fundamental transformation is named λ-representation as listed below.

$$f(x) = \sum_{h \in \mathcal{P}} f(h) \lambda_h \tag{23}$$

$$\sum_{h \in \mathcal{P}} h \lambda_h = x \tag{24}$$

$$\sum_{h \in \mathcal{P}} \lambda_h = 1 \tag{25}$$

$$\forall \lambda_h \in R^+, \forall h \in \mathcal{P} \tag{26}$$

where $\mathcal{P}$ is the set that consists of all the possible values of x. Therefore in our case, $\mathcal{P} = \{0, 1\}$. As we can see that, it introduces $|\mathcal{P}|$ extra variables λ_h in the transformation and makes the original function to be a new function over λ_h and x. As indicated in the formulation, λ_h could be any positive real numbers and x equals the weighted combination of λ_h. By applying λ-representation to (17), we can easily get the new form of our problem formulation, which is an LP as listed below:

$$\min_{X, \boldsymbol{\lambda}} \quad \sum_{i=1}^{n} \sum_{j=1}^{d} \Big(\sum_{h \in \mathcal{P}} k^{(c_{ij} + e_{ij}) \cdot h} \lambda_{ij}^{h} \Big) \tag{27}$$

$$\text{s.t.} \quad \sum_{h \in \mathcal{P}} h \lambda_{ij}^{h} = x_{ij},\ \forall i \in \mathcal{N},\ \forall j \in \mathcal{D} \tag{28}$$

$$\sum_{h \in \mathcal{P}} \lambda_{ij}^{h} = 1, \qquad \forall i \in \mathcal{N},\ \forall j \in \mathcal{D} \tag{29}$$

$$\lambda_{ij}^{h} \in R^+, \qquad \forall i \in \mathcal{N},\ \forall j \in \mathcal{D},\ \forall h \in \mathcal{P} \tag{30}$$

$$(1), (2), (3), (4).$$

As $\mathcal{P} = \{0, 1\}$, we can further expand and simplify the above formulation to get our final formulation as follows:

$$\min_{X, \boldsymbol{\lambda}} \quad \sum_{i=1}^{n} \sum_{j=1}^{d} \big((k^{(c_{ij} + e_{ij})} - 1) \cdot \lambda_{ij}^{1} \big) \tag{31}$$

$$\text{s.t.} \quad \lambda_{ij}^{1} = x_{ij},\ \forall i \in \mathcal{N},\ \forall j \in \mathcal{D} \tag{32}$$

$$(1), (2), (3), (4), (30).$$

We can clearly see that it is an LP with only nd variables, where n is the number of tasks and d is the number of data centers. As it is an LP over X, it can be efficiently solved by standard linear programming solvers like Breeze [18] in Scala [19], and because the coefficients in the constraints form a totally unimodular matrix, its optimal solutions for X are integral and the same as the solutions to the original ILP problem **P1**.

3.2 Workloads with Network Budget Constraints

For the workloads with network budget constraints, we are not able to transform the problem **P2** to a linear programming problem. In our implementation, we solve the ILP problem **P2** directly to obtain the scheduling results. Later we will show that we can achieve near optimal job completion time and network cost even when $\theta = 1.0$, which means that we can directly calculate the scheduling results by solving the LP problem in Eq. (7) instead of solving the ILP problem in **P2**. We will illustrate the details in the experimental results.

4 Design and Implementation

After we have discussed how our task scheduling problem can be solved efficiently, we are now ready to see how we implement it in Spark, a modern framework popular for big data processing.

Spark is a fast and general distributed data analysis framework. Different from disk-based Hadoop [20], Spark would cache a part of the intermediate results in memory. Thus it would greatly speed up iterative jobs as it can directly obtain the outputs of the previous stage from main memory instead of the disk. Now as Spark becomes more and more mature, several projects designed for different applications are built upon Spark such as `MLlib`, Spark Streaming and Spark SQL. All these projects rely on the core module of Spark, which contains several fundamental functionalities of Spark including *Resilient Distributed Datasets* (RDDs) and scheduling.

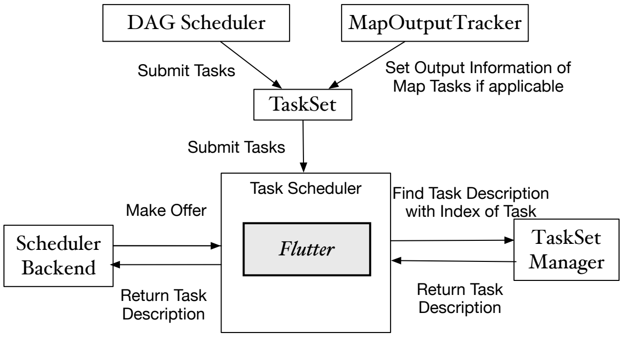


Fig. 3: The design of *Flutter* in Spark.

To incorporate our scheduling algorithm in Spark, we override the scheduling modules to implement our algorithm. From the top of the view, after a job is submitted in Spark, the job would be transformed into a DAG of tasks and handled by the DAG scheduler. Then, the DAG scheduler would first check whether the parent stages of the final stage are complete. If they are, the final stage is directly submitted to the task scheduler for the task scheduling. If not, the parent stages of the final stage are submitted recursively until the DAG scheduler finds a ready stage.

The detailed architecture of our implementation can be seen in Fig. 3. As we can observe from the figure, after the DAG scheduler finds a ready stage, it would create a new `TaskSet` for that ready stage. Here if the `TaskSet` is a set of reduce tasks, we would first get the output information of the map tasks from the `MapOutputTracker`, and then save it to this `TaskSet`. Then this `TaskSet` would be submitted to the task scheduler and added to a list of pending `TaskSets`. When the `TaskSets` are waiting for resources, the `SchedulerBackend`, which is also the cluster manager, would offer some free resources in the cluster. After receiving the resources, *Flutter* would pick a `TaskSet` in the queue, and determine which task should be assigned to which executor. It also needs to interact with `TaskSetManager` to obtain the description of the tasks, and later return these task descriptions to the `SchedulerBackend` for launching the tasks. During the entire process, getting the outputs of the map tasks and the scheduling process are the two key steps; in what follows, we will present more details about these two steps.

4.1 Obtaining Outputs of the Map Tasks

Flutter needs to compute the transfer time/cost to obtain all the intermediate results for each reduce task if it is scheduled to a data center. Therefore, obtaining the information about the outputs of map tasks including both the locations and the sizes is a key step towards our goal. Here we will first introduce how we obtain the information about the map outputs.

A `MapOutputTracker` is designed in the driver of Spark to let reduce tasks know where to fetch the outputs of the map tasks. It works as follows. Each time when a map task finishes, it would register the sizes and the locations of its outputs to the `MapOutputTracker` in the driver. Then if the reduce tasks want to know the locations of the map outputs, it will send messages to the `MapOutputTracker` directly to get the information.

In our case, we can obtain the output information of map tasks in the DAG scheduler through the `MapOutputTracker`, as the map tasks have already registered its output information to the `MapOutputTracker`. We then save the output information of map tasks to the `TaskSet` of reduce tasks before submitting the `TaskSet` to the task scheduler. Therefore the `TaskSet` would carry the output information of the map tasks and be submitted to the task scheduler for task scheduling.

4.2 Task Scheduling with Flutter

The task scheduler serves as a ”bridge” that connects tasks and resources (executors in Spark). On the one hand, it will keep receiving `TaskSets` from the DAG scheduler. On the other hand, it would be notified if there are newly available resources by the `SchedulerBackend`. For instance, each time when a new executor joins the cluster or an executor has finished one task, it would offer its resources along with its hardware specifications to the task scheduler. Usually, multiple offers from several executors would reach the task scheduler at the same time. After receiving these resource offers, the task scheduler then starts to use its scheduling algorithm to the pick up the right pending tasks that are most suited to the offered resources.

In our task scheduling algorithm, after we receive the resource offers, we first pick a `TaskSet` in the sorted list of `TaskSets` and check whether it has shuffle dependency. In other words, we want to check whether tasks in this `TaskSet` are reduce tasks. If they are, we need to do two things. The first is to get the output information of the map tasks and calculate the transfer times for each possible scheduling decision. We do not consider the execution times of the tasks in the implementation because the execution times of the tasks in a stage are almost uniform on homogeneous VMs, which will not affect the scheduling results as our problem is a min-max problem. Similar strategy is also adopted in [4], [21]. The second is to figure out the amount of available resources on each data center through received resource offers. After these two steps, we feed the information to our linear programming solver, and the solver would return an index of the most suitable data center for each reduce task. Finally, we randomly choose a host that has enough resource for the task on that data center and return the task description to `SchedulerBackend` for launching the task. If the `TaskSet` does not have shuffle dependency, the default delay scheduling [22] would be adopted. Thus each time, when there are new resource offers, and the pending `TaskSet` is a set of reduce tasks, *Flutter* would be invoked. Otherwise, the default scheduling strategy is used.

For workloads with network budget constraints, the whole procedure is similar. The differences exist in two aspects. First, in this case, we need to calculate the potential cost incurred by the data transfers for each possible scheduling. Second, we need to solve the optimization problem to obtain the optimal bandwidth cost, which would be fed to the final optimization problem for scheduling decisions. After we decide which data center to place each task, other details are same with the case for workloads without network budget constraints as we stated above.

5 Performance Evaluation

In this section, we will present our experimental setup in geo-distributed data centers and detailed experiment results on real-world workloads.

5.1 Experimental Setup

We first describe the testbed we used in our experiments, and then briefly introduce the applications, baselines, and metrics used throughout the evaluations.

Private Testbed: Our experiments are conducted in 6 data centers with a total of 25 instances, among which two data centers are in Toronto. The other data centers are located at various academic institutions: Victoria, Carleton, Calgary and York. All the instances used in the experiments are m.large, which has four cores and 8 GB of main memory. The bandwidth capacities among VMs in these regions are measured by `iperf` and are shown in Table 4. The data centers in Ontario are interconnected through dedicated 1GE links. Hence we can see in the table that the bandwidth capacities between the data centers in Toronto, Carleton and York are relatively high, while they are still lower than the bandwidth capacities within the same data center.

TABLE 4: Available bandwidths between VMs across geo-distributed data centers as of July 2015 (Mbps).

	Tor-1	Tor-2	Victoria	Carleton	Calgary	York
Tor-1	1000	931	376	822	99.5	677
Tor-2	-	1000	389	935	97.1	672
Victoria	-	-	1000	381	82.5	408
Carleton	-	-	-	1000	93.7	628
Calgary	-	-	-	-	1000	95.6
York	-	-	-	-	-	1000

Note: "Tor" is short for Toronto. Tor-1 and Tor-2 are two data centers located at Toronto.

EC2 Deployment: Our cluster on Amazon EC2 consists of 16 instances across five regions (N. Virginia, N. California, Frankfurt, Singapore and Sydney). All the instances are m3.xlarge and each instance has four vCPUs and 15GB of main memory. The bandwidths between VMs across regions can be found in Table 5. The pricing (USD/GB) for sending data across regions in those 5 regions are 0.02, 0.02, 0.02, 0.09 and 0.16 respectively. As we can see that, the bandwidth cost in Sydney could be 8 times to the cost of the regions in the US and Europe.

TABLE 5: Available bandwidths between VMs across geo-distributed data centers on EC2 as of Oct. 2016 (Mbps).

	US-East	US-West	EU	Singapore	Sydney
US-East	1105.9	90.0	67.1	21.2	14.8
US-West	-	1105.9	69.2	38.2	72.6
EU	-	-	1105.9	28.4	31.8
Singapore	-	-	-	1105.9	64.7
Sydney	-	-	-	-	1105.9

HDFS: The distributed file system used in our geo-distributed cluster is the Hadoop Distributed File System (HDFS) [20]. We use one instance as the master node for both HDFS and Spark. All the other nodes are served as `datanodes` and worker nodes. The block size in HDFS is 128MB, and the number of replications is 3.

Applications: We deploy three applications on Spark. They are WordCount, PageRank [23] and GraphX [24].

- **WordCount**: WordCount calculates the frequency of every single word appearing in a single or batch of files. It would first calculate the frequency of words in each partition, and then aggregate the results in the previous step to obtain the final result. We choose WordCount because it is a fundamental application in distributed data processing and can process the real-world data traces such as Wikipedia dumps.
- **PageRank**: It computes the weights for websites based on the amount and quality of links that point to the websites. This method relies on the assumption that a website is important if many other important websites are linking to it. It is a typical data processing application with multiple iterations. We use it for calculating both the ranks for the websites and the impact of users in social networks.
- **GraphX**: GraphX is a module built upon Spark for parallel graph processing. We run the application *LiveJournalPageRank* as the representative application on top of GraphX. Even though the application is also named "PageRank," the computation module is completely different on GraphX. We choose it because we also wish to evaluate *Flutter* on systems built upon Spark.

Inputs: For WordCount, we use 10GB of Wikipedia dump as the input. For PageRank, we use an unstructured graph with 875713 nodes and 5105039 edges released by Google [25] in the private testbed and a directed graph with 1632803 nodes and 30622564 edges from Pokec online social network [26] in the case of EC2 deployment. For GraphX, we adopt a directed graph in *LiveJournal* online social network with 4847571 nodes and 68993773 edges [25], where *LiveJournal* is a free online community.

Baseline: In the private testbed, we compare our task scheduler with delay scheduling [22], which is the default task scheduler in Spark. In the Amazon EC2 deployment, we also compare our scheduler with Iridium [4] [3]. In this case, we denote flutter for workloads without network budget constraints as *Flutter w/o Cost Awareness* and the other algorithm as *Flutter w/ Cost Awareness*.

Metrics: The first two metrics used are job completion times and stage completion times of the three application. As the bandwidths among different data centers are expensive regarding cost, so we also take the amount of traffic transferred among different data centers and the costs incurred by the data transfers as two other metrics in the Amazon EC2. Moreover, we also report the running times of solving the LP in different scales to show the scalability of our approach.

5.2 Results on the Testbed

In the experiments on the testbed, we examine the performance of Flutter for workloads without network budget constraints. Here we wish to first answer the following questions. (1) What are the benefits of *Flutter* regarding job

[3] We implement the LP problem for the reduce task scheduling in this paper.

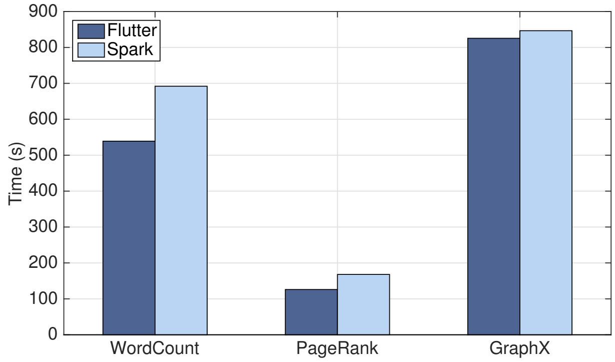
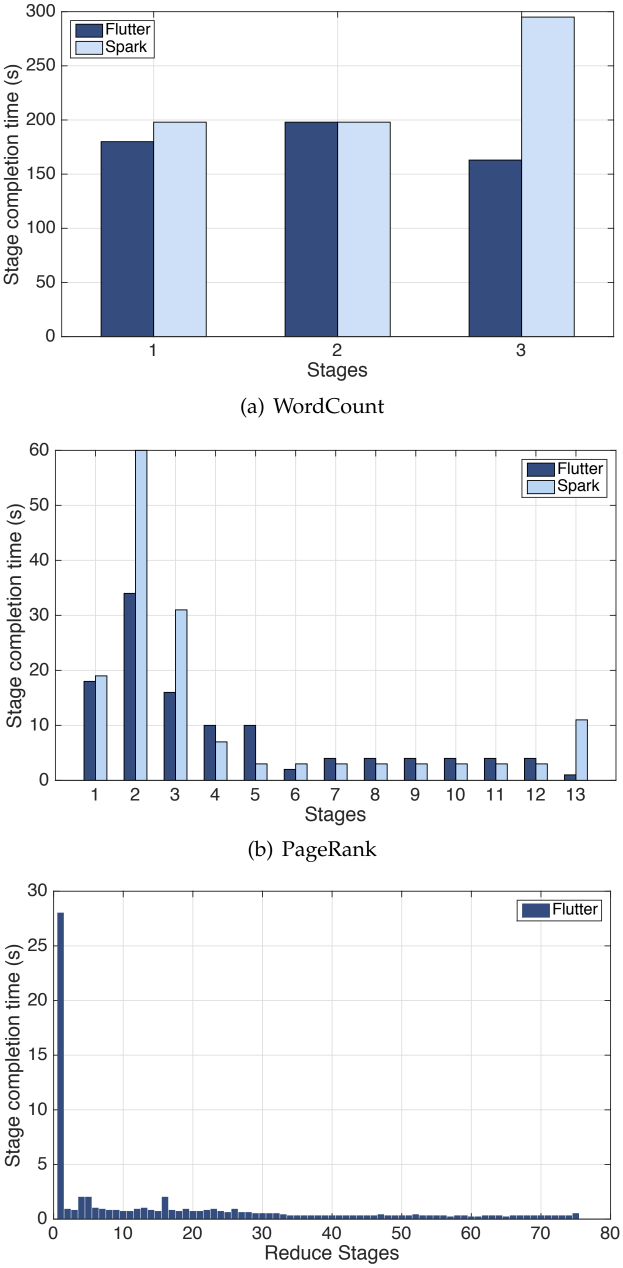


Fig. 4: The job computation times of the three workloads.



(a) WordCount

(b) PageRank

(c) GraphX

Fig. 5: The completion times of stages in WordCount, PageRank and GraphX.

completion times, stage completion times, as well as the volume of data transferred among different data centers? (2) Is *Flutter* scalable regarding the times to compute the scheduling results, especially for short-running tasks?

5.2.1 Job Completion Times

We plot the job completion times of the three applications in Fig. 4. As we can see, completion times of all three applications with *Flutter* have been reduced. More specifically, *Flutter* reduced the job completion time of WordCount and PageRank by 22.1% and 25%, respectively. The completion time of GraphX is also reduced by more than 20 seconds. There are primarily two reasons for the improvements. The first is that *Flutter* can adaptively schedule the reduce tasks to a data center that would cost the least amount of transfer times to get all the intermediate results. Thus it can start the tasks as soon as possible. The second is that *Flutter* would schedule the tasks in the stage as a whole. Therefore, it can significantly mitigate the stragglers — the slow-running tasks in that stage — and further improve the overall performance.

It seems that the improvements regarding job completion times on GraphX are small in this case, which is because there are three data centers that have high bandwidths with each other and delay scheduling may also schedule the tasks in those three data centers. Even though the job completion time is not reduced significantly for GraphX applications, we will show that *Flutter* would significantly reduce the amount of traffic transferred across different data centers for GraphX applications.

5.2.2 Stage Completion Times

As *Flutter* schedules the tasks stage by stage, we also plot the completion times of stages in these applications in Fig. 5. In this way, we can have a closer view of the scheduling performance of both our approach and the default scheduler in Spark, by checking the performance gap stage by stage and finding out how the overall improvements of job completion times are achieved. We will explain the performance of the three applications one by one.

For WordCount, we repartition the input datasets as the input size is large. Therefore it has three stages: repartition, reduceByKey (map), and reduce. In the first stage, as it is not a stage with shuffle dependency, we use the default scheduler in Spark. Thus the performance achieved is almost the same. In the second stage, the map stage is normally not a stage with shuffle dependency. However, as it runs after repartition stage, it becomes a stage with shuffle dependency. We can see that the stage completion times of this stage for the two schedulers are the same, which is because the default scheduler also schedules the tasks in the same data centers as ours. In the reduce stage, which is also the last stage, our approach takes only 163 seconds, while the default scheduler in Spark takes 295 seconds, which is almost twice as long. The performance improvements are due to both network-awareness and stage awareness, as *Flutter* schedules the tasks in that stage as a whole, and take the transfer times into consideration at the same time. It

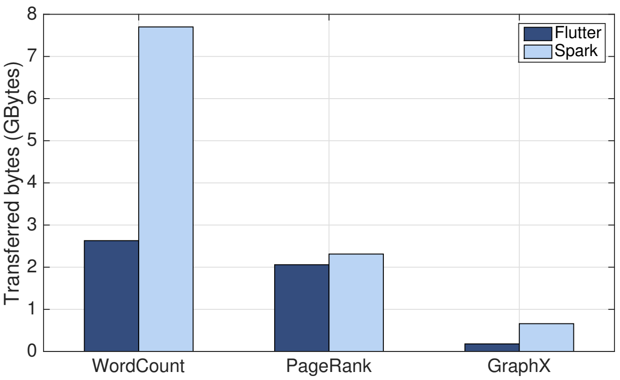


Fig. 6: The amount of data transferred among different data centers.

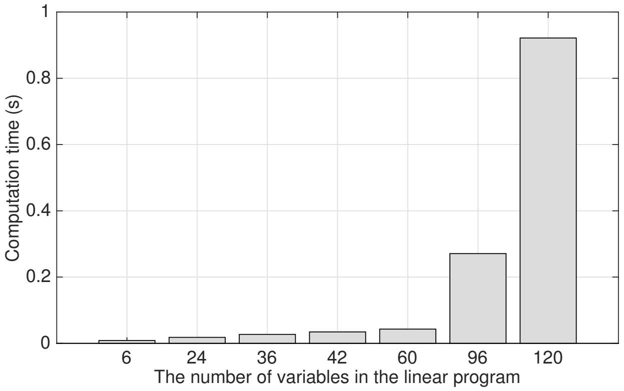


Fig. 7: The computation times of *Flutter*'s linear program at different scales.

can effectively reduce the number of straggler tasks and the transfer times to get all the inputs.

We draw the stage completion times of PageRank in Fig. 5(b). As we can see in this figure, it has 13 stages in total, including two *distinct stages*, 10 *reduceByKey stages* and one *collect stage* to collect the final results. We have 10 *reduceByKey* stage because the number of iterations is 10. Except for the first *distinct stage*, all the other stages are shuffle dependent. So we adopt *Flutter* instead of delay scheduling for task scheduling in those stages. As we can see in stage 2, 3 and 13, we have far shorter stage completion times compared with the default scheduler. Especially in the last stage, *Flutter* takes only 1 second to finish that stage, while the default scheduler takes 11 seconds.

Fig. 5(c) depicts the completion times of reduce stages in GraphX. As the total number of stages is more than 300, we only draw the reduce stages in that job. Because the stage completion times of these two schedulers are similar, we only draw the stage completion time of *Flutter* to illustrate the performance of GraphX. First, we can see that the first reduce stage took about 28 seconds, while the following reduce stages completed quickly, which takes only 0.4 seconds. We can see that GraphX completes quickly for later stages.

5.2.3 Data Volume Transferred across Data Centers

After we see the improvements of job completion times, we are now ready to evaluate the performance of *Flutter* regarding the amount of data transferred across geo-distributed data centers in Fig. 6. In WordCount, the amount of data transferred across different data centers with the default scheduler is around three times to the one of *Flutter*. The amount of data across data centers when running GraphX is four times to our approach. In the case of PageRank, we also achieved lower volumes of data transfers.

Even though reducing the amount of data transferred across different data centers is not the main goal of our optimization, we find out that it is in line with the goal of reducing the job completion time for data processing applications on distributed data centers. This is because the bandwidth capacities across VMs in the same data center are higher than those on inter-data center links. When *Flutter* tries to place the tasks to reduce the transfer times to get all the inputs, it will prefer to put the tasks in the data center that has most of the input data. Thus, it can reduce the volume of data transfers across different data centers by a substantial margin.

5.2.4 Scalability

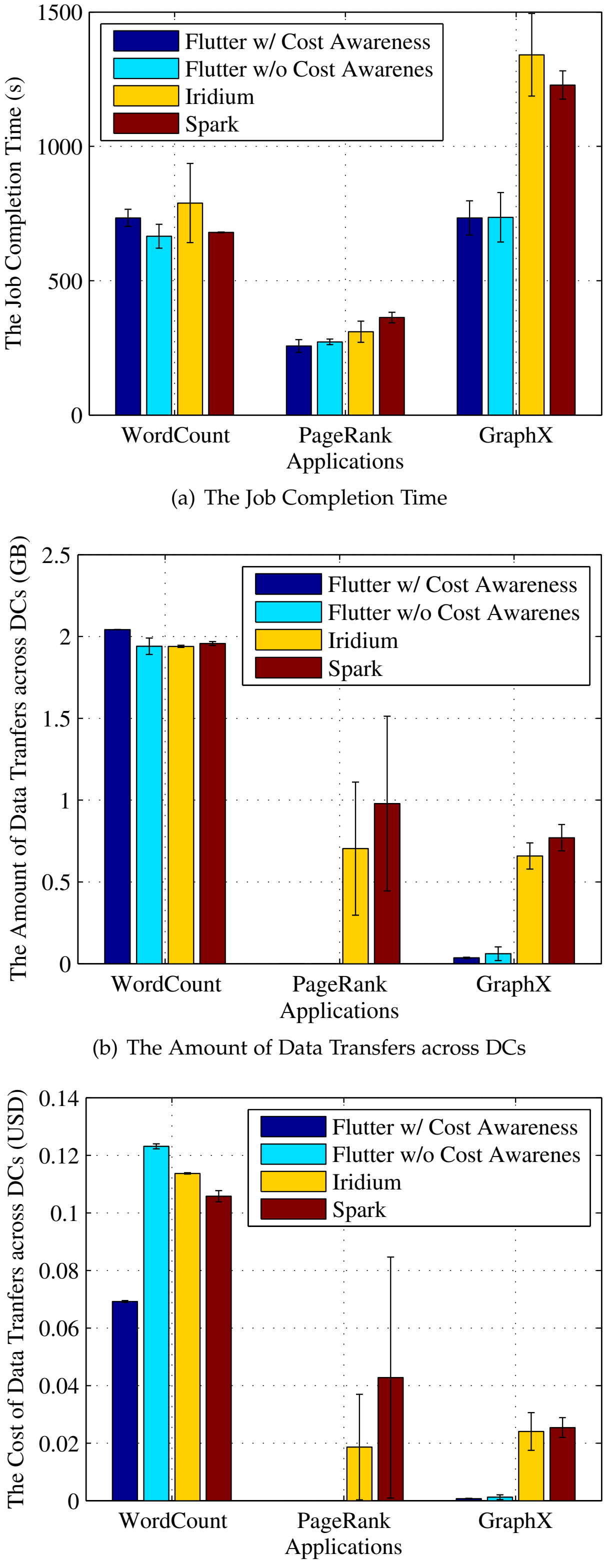
Practicality is one of the main objectives when designing *Flutter*, which means that *Flutter* needs to be efficient at runtime. Therefore, we record the time it takes to solve the LP (**P1**) when we run Spark applications. The results have been shown in Fig. 7. In the figure, the number of variables varies from 6 to 120 and the computation times are averaged over multiple runs. We can see that the linear program is rather efficient: it takes less than 0.1 second to return the result for 60 variables. Moreover, the computation time is less than 1 second for 120 variables, which is also acceptable because the transfer times could be tens of seconds across distributed data centers. *Flutter* is scalable because it is formulated as an efficient LP, which can be solved efficiently by standard LP solvers. Moreover, we can further reduce the solver latency by using commercial LP solvers like CPLEX [27] and Mosek [28], which can return the solutions for problems with thousands of variables within one second.

5.3 Results on EC2

After reviewing the results in the private testbed, here we further examine the performance of the four algorithms including flutter with/without network budget constraints, Iridium and Spark (delay scheduling) on Amazon EC2. First, we want to know the performance of these algorithms regarding job completion time, the amount of data transfers and the costs incurred by those data transfers. Second, we will see how the performance of flutter with network budget constraints varies with θ.

5.3.1 Time, Data and Cost

We conduct all the experiments for five rounds. The default parallelism in WordCount is configured to be 12. For PageRank, the number of iterations is 10, and the number of partitions in GraphX is set to be 12. The number of CPU cores for each task is 1 for all the three applications. The value of θ is 1.0. With these settings, the results are shown in Fig. 8. In Fig. 8(a), it shows that the job completion times of the three applications. In this figure, we can see



(a) The Job Completion Time

(b) The Amount of Data Transfers across DCs

(c) The Cost of Data Tranfers across DCs

Fig. 8: The completion times, amount of data transfers across DCs and the costs incurred by the data transfers.

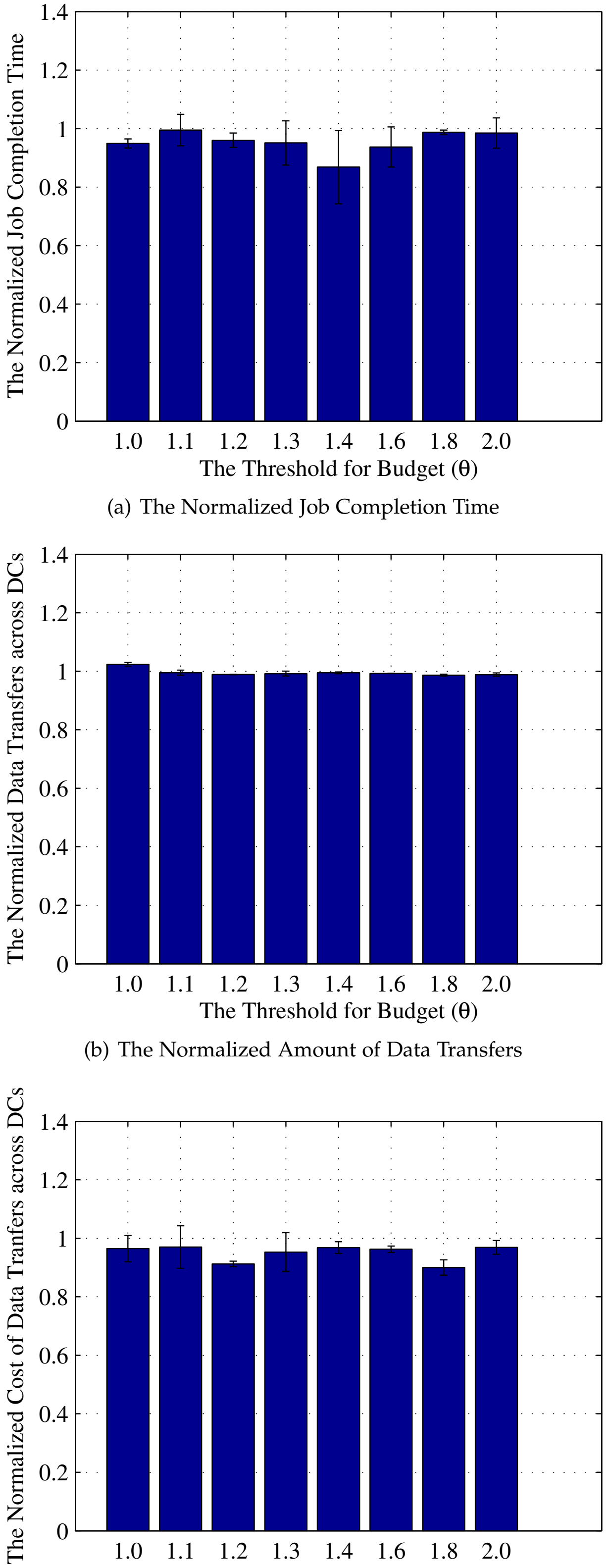
that two algorithms of Flutter perform similar with Iridium and Spark for WordCount, and Iridium performs the worst. In PageRank, our algorithms have lower job completion times, and Spark is the worst instead. In GraphX, our two algorithms show remarkable improvements over the two state-of-art algorithms. The reason for the results can be well explained by the amount of data transferred across different data centers.

Now we present the amount of data transferred across data centers in Fig. 8(b). In this figure, we can see that the amount of data transferred across data centers are similar for WordCount. However, both of our algorithms can substantially reduce the amount of data transferred for PageRank and GraphX. More specifically, in PageRank, the amount of data transferred across data centers of our algorithms is 0 while the mean value for Iridium and Spark is around 0.75 GB and 1 GB. The reason is that the inputs for PageRank are all in the same data center, and our algorithms schedules all the following tasks in that data center. However, Iridium may schedule the tasks to other data centers for the rounding process because the original solution may not always be integral. For delay scheduling in Spark, it incurs more traffic across data centers because it does not consider the sizes and locations of intermediate results at all. The above reasons also explain the results of GraphX. But as the inputs of GraphX are in different data centers, data transfers across data centers are inevitable, which results in a small amount of data transferred for our two algorithms.

Besides the job completion time and the amount of data transferred across data centers, we also investigate the costs incurred by the data transfers across data centers in Fig. 8(c). In this figure, we can clearly see that the shape is very much alike the one in Fig. 8(b). More specifically, for PageRank and GraphX, we have much less amount of data transfers in Fig. 8(b) and we can see that we have much lower cost in Fig. 8(c). The most interesting part is in the case of WordCount. In Fig. 8(b), it shows that the amount of data transferred across data centers for flutter with cost awareness is slightly higher than the other three. However, in Fig. 8(c), the cost incurred by those data transfers is only around half of the ones of other three algorithms, which means that flutter with cost awareness is actually working and can substantially reduce the cost incurred by the data transfers across data centers. At the same time, the performance of job completion time is also quite good as shown in Fig. 8(a).

5.3.2 The Effect of θ

In the formulation of flutter with cost awareness, we use θ as the tunable parameter to adjust the budget of network cost. In prior experiments, θ is set to be 1.0, which means that the solution is the same with the one that aims to minimize the total network cost as shown in Eq. (7). We can also see that in prior results, limiting the network budget would not harm the job completion time but instead reduce the cost of data transfers effectively. Here we set the number of CPU cores needed for each task to be 4 to distribute the tasks to as many data centers as possible and θ from 1.0 to 2.0 to present more results about the flutter with cost awareness. We conduct each experiment for several rounds.



(a) The Normalized Job Completion Time

(b) The Normalized Amount of Data Transfers

(c) The Normalized Cost of Data Transfers

Fig. 9: The normalized completion times, amount of data transfers across DCs and the costs incurred by the data transfers w.r.t. Iridium.

Here we show how the performance would vary with θ in Fig. 9. The results are normalized with respect to Iridium. In Fig. 9(a), we outperform Iridium almost in all the cases and reduce the job completion time by up to 16%. In Fig. 9(b), we can see that the amount of data transferred across data centers are almost the same; however, we can effectively reduce the cost incurred by up to 10%. The results also imply that directly setting the θ to be 1.0 can yield good scheduling results already, which is also shown in Fig. 8. In other words, we can directly solve the linear programming problem in Eq. (7) to generate the scheduling results. Note that the improvements are narrowed in this figure as compared with Fig. 8, which is because the number of cores needed for each task is 4 and it will decrease the number of tasks that can be scheduled in each data center.

5.3.3 ILP Execution Time

Here we also present the computation times of the integer linear programming problem in Eq. (9). The maximum number of variables is 60. We record all the computation times for the ILP at the runtime. We can see that the average times are around 0.04 seconds and the standard deviations are also less than 0.083 for all the three number of variables. Therefore, it shows that the integer linear programming problem is also very efficient at this scale.

TABLE 6: The computation times of the integer linear programming problem in Eq. (9) (s).

Number of Variables	50	55	60
Average Time	0.0479	0.0494	0.0406
Standard Deviation	0.0800	0.0820	0.0723

6 Related Work

In this section, we first show a few most related work in geo-distributed big data processing and geo-distributed storage services. We then survey some scheduling systems in distributed data processing systems in a single data center.

Geo-distributed data analytics has been studied in [5], [21], [6], [29], [30], [31], [32]. In [5], the authors design an integer programming problem for optimizing the query execution plan and the data replication strategy to reduce the bandwidth costs. As they assume each data center has unlimited storage, they aggressively cache the results of previous queries to reduce the data transfers of subsequent queries. Clarinet [21] is also designed for SQL queries. It feeds the bandwidth information to the query plan optimizer to generate better query plans for queries over geo-distributed data. Geo-distributed big data processing over general big data processing systems like Hadopp and Spark are discussed in Pixida [6] and SWAG [30]. In Pixida [6], they propose a new way to aggregate the tasks in the original DAG to make the DAG simpler. After that, they propose a new generalized min-k-cut algorithm to divide the simplified DAG into several parts for execution, and each part would be executed in one data center. The case of scheduling multiple jobs across geo-distributed data centers is discussed in SWAG [30]. The aforementioned approaches are all for general big data processing applications or SQL queries. Geo-distributed machine learning is addressed in [31], [32]. GDML [31] is built upon YARN and designed to accelerate the machine learning applications. However, Gaia [32] improves the performance by designing a new synchronization model, which avoids unnecessary global synchronizations and thus saves the traffic across geo-distributed data centers.

The most related recent work is Iridium [4] for low latency geo-distributed analysis, while we have some significant differences with it. First, they assume the network connecting the *sites* (data centers) are congestion-free and the network bottlenecks only exist in the up/down links of VMs. This is not the case in our measurements. In our measurements, the in/out bandwidths of VMs are both 1 Gbps in intra-data centers, while the bandwidths among VMs in different data centers are only around 100 Mbps. Therefore the network bottlenecks are more likely to exist in the network connecting the data centers instead. Second, in their linear programming formulation for task scheduling, they assume reduce tasks are infinitesimally divisible and each reduce task would receive the same amount of intermediate results from the map tasks, which are unrealistic assumptions as reduce tasks are not divisible with low overhead and the data skews are common in the data processing frameworks [33]. While we use the exact amount of intermediate results that each reduce task would read from the outputs of map tasks.

Besides job scheduling across geo-distributed data centers, geo-distributed storage services and load balancing problems are investigated in [34], [35], [36]. More specifically, a carbon-aware geo-distributed storage system is proposed in [34]. Instead, the authors in [35] aim to minimize the monetary costs of instances including both on-demand and spot instances for in-memory storage workloads. Our work is different from these two systems because we focus on the task scheduling problem given the fixed input data placements and we target at big data processing applications on top of Spark or Hadoop. While we focus on the task scheduling problem in the task-level, a service-level geographical load balancing scheme to minimize the cost including energy cost and delay cost is proposed in [36].

General task/job scheduling in data processing systems has been investigated in [37], [38], [39], [40], [41], [42], [43]. More specifically, Yarn [37] and Mesos [38] are the cluster managers designed for improving cluster utilization. Sparrow [39] is a decentralized scheduling system for Spark that can schedule a significant number of jobs at the same time with little scheduling delays, and Hopper [40] is a unified speculation-aware scheduling framework for both centralized and decentralized schedulers. Moreover, HUG [42] is designed to achieve multi-resource fairness for elastic and correlated demands.

7 Concluding Remarks

In this paper, we focus on how tasks may be scheduled closer to the data across geo-distributed data centers for workloads with/without network budget constraints. We first find out that the network could be a bottleneck for geo-distributed big data processing, by measuring available bandwidth across Amazon EC2 data centers. Thus we formulate our problem for workloads without network budget constraints as an integer linear programming problem, considering both the network and the computational resource constraints. We also find out that we can transform the integer linear programming problems into a linear programming problem, with the same optimal solution. However, we identify that achieving the optimal completion time would not guarantee the optimal network cost. Therefore we formulate the problem for workloads with network budget constraints separately as another integer linear programming problem.

Based on these theoretical insights, we have designed and implemented *Flutter*, a new framework for scheduling tasks across geo-distributed data centers for both workloads with/without network budget constraints. With real-world performance evaluations using an inter-data center network testbed and VMs on Amazon EC2, we have shown convincing evidence that *Flutter* is not only able to shorten the job completion times but also to reduce the amount of traffic that needs to be transferred to other data centers. We can also substantially lower the cost incurred by the data transfers across data centers. In our future work, we will investigate how data placement, replication, and task scheduling can be jointly optimized for even better performance in the context of wide-area big data processing. We also plan to apply the DAG scheduling techniques to directly optimize the job level performance.

References

[1] Z. Hu, B. Li, and J. Luo, “Flutter: Scheduling Tasks Closer to Data across Geo-distributed Datacenters,” in *Proc. IEEE INFOCOM*, 2016.

[2] J. J. Stephen, D. Gmach, R. Block, A. Madan, and A. AuYoung, “Distributed Real-Time Event Analysis,” in *Proc. IEEE ICAC*, 2015.

[3] “How to Fight the New Breed of DDoS Attacks on Data Centers,” "https://goo.gl/5xX7eH", accessed on 02.11.2017.

[4] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica, “Low Latency Geo-Distributed Data Analytics,” in *Proc. ACM SIGCOMM*, 2015.

[5] A. Vulimiri, C. Curino, B. Godfrey, J. Padhye, and G. Varghese, “Global Analytics in the Face of Bandwidth and Regulatory Constraints,” in *Proc. USENIX NSDI*, 2015.

[6] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, “Pixida: Optimizing Data Parallel Jobs in Bandwidth-Skewed Environments,” in *Proc. VLDB*, 2015.

[7] Y.-K. Kwok and I. Ahmad, “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors,” *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[8] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, and V. ICSI, “Making Sense of Performance in Data Analytics Frameworks,” in *Proc. of USENIX NSDI*, 2015.

[9] “Microsoft Azure: Cloud Computing Platform & Services.” [Online]. Available: https://azure.microsoft.com/en-us/

[10] “Amazon EC2.” [Online]. Available: http://aws.amazon.com/ec2/

[11] “Google Compute Engine.” [Online]. Available: https://cloud.google.com/compute/

[12] “Microsoft Azure: Data Transfers Pricing Details.” [Online]. Available: https://goo.gl/e1iPc9

[13] “EC2 Instance Pricing - Amazon Web Services (AWS).” [Online]. Available: https://aws.amazon.com/ec2/pricing/

[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing,” in *Proc. USENIX NSDI*, 2012.

[15] M. Mao and M. Humphrey, “A Performance Study on the VM Startup Time in the Cloud,” in *Proc. IEEE CLOUD*, 2012.

[16] 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.

[17] 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 *Proc. IEEE INFOCOM*, 2015.

[18] "Breeze: a numerical processing library for Scala." [Online]. Available: https://github.com/scalanlp/breeze.

[19] "Scala." [Online]. Available: http://www.scala-lang.org/

[20] "Hadoop." [Online]. Available: https://hadoop.apache.org/

[21] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "CLARINET: WAN-Aware Optimization for Analytics Queries," in *Proc. USENIX OSDI*, 2016.

[22] 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. of ACM Eurosys*, 2010.

[23] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web." 1999.

[24] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph Processing in a Distributed Dataflow Framework," in *Proc. USENIX OSDI*, 2014.

[25] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[26] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, Jun. 2014.

[27] "IBM ILOG CPLEX Optimizer." [Online]. Available: https://goo.gl/jyvDuV

[28] "Mosek." [Online]. Available: https://www.mosek.com/

[29] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese, "WANalytics: Analytics for a Geo-distributed Data-intensive World," in *Proc. CIDR*, 2015.

[30] C.-C. Hung, L. Golubchik, and M. Yu, "Scheduling Jobs Across Geo-distributed Datacenters," in *Proc. ACM SoCC*, 2015.

[31] I. Cano, M. Weimer, D. Mahajan, C. Curino, and G. M. Fumarola, "Towards Geo-Distributed Machine Learning," *arXiv preprint arXiv:1603.09035*, 2016.

[32] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds," in *Proc. USENIX NSDI*, 2017.

[33] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proc. ACM SIGMOD*, 2012.

[34] Z. Xu, N. Deng, C. Stewart, and X. Wang, "Cadre: Carbon-Aware Data Replication for Geo-Diverse Services," in *Proc. IEEE ICAC*, 2015.

[35] Z. Xu, C. Stewart, N. Deng, and X. Wang, "Blending On-Demand and Spot Instances to Lower Costs for In-Memory Storage," in *Proc. IEEE INFOCOM*, 2016.

[36] Z. Liu, M. Lin, A. Wierman, S. Low, and L. L. Andrew, "Greening Geographical Load Balancing," *IEEE/ACM Transactions on Networking (TON)*, vol. 23, no. 2, pp. 657–671, 2015.

[37] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache Hadoop Yarn: Yet Another Resource Negotiator," in *Proc. ACM SoCC*, 2013.

[38] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proc. USENIX NSDI*, 2011.

[39] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proc. ACM SOSP*, 2013.

[40] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu, "Hopper: Decentralized Speculation-aware Cluster Scheduling at Scale," in *Proc. ACM SIGCOMM*, 2015.

[41] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *Proc. ACM SIGOPS*, 2009.

[42] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-Resource Fairness for Correlated and Elastic Demands," in *Proc. USENIX NSDI*, 2016.

[43] Z. Hu, B. Li, Z. Qin, and R. S. M. Goh, "Job Scheduling without Prior Information in Big Data Processing Systems," in *Proc. IEEE ICDCS*, 2017.

Zhiming Hu received his BS degree in computer science from Zhejiang University, China, in 2011 and his Ph.D. degree from the School of Computer Science and Engineering, Nanyang Technological University, Singapore, in 2016. He is now a postdoctoral fellow in the Department of Electrical and Computer Engineering, University of Toronto, Canada. His research interests include big data processing, data center networking, and cloud computing.

Baochun Li received his B.Engr. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 1995 and his M.S. and Ph.D. degrees from the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, in 1997 and 2000.

Since 2000, 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 cloud computing, multimedia systems, applications of network coding, and wireless networks. He was the recipient of the IEEE Communications Society Leonard G. Abraham Award in the Field of Communications Systems in 2000, the Multimedia Communications Best Paper Award from the IEEE Communications Society in 2009, and the University of Toronto McLean Award in 2009. He is a member of ACM and a Fellow of IEEE.

Jun Luo received his BS and MS degrees in Electrical Engineering from Tsinghua University, China, and the Ph.D. degree in Computer Science from EPFL (Swiss Federal Institute of Technology in Lausanne), Lausanne, Switzerland. From 2006 to 2008, he has worked as a postdoctoral research fellow in the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada. In 2008, he joined the faculty of the School of Computer Engineering, Nanyang Technological University in Singapore, where he is currently an associate professor. His research interests include wireless networking, mobile and pervasive computing, applied operations research, as well as network security. More information can be found at http://www3.ntu.edu.sg/home/junluo.