Flutter: Scheduling Tasks Closer to Data Across Geo-Distributed Datacenters

Zhiming Hu[1], Baochun Li[2], and Jun Luo[1]

[1]School of Computer Engineering, Nanyang Technological University, Singapore, {zhu007, junluo}@ntu.edu.sg
[2]Department of Electrical and Computer Engineering, University of Toronto, Canada, bli@ece.toronto.edu

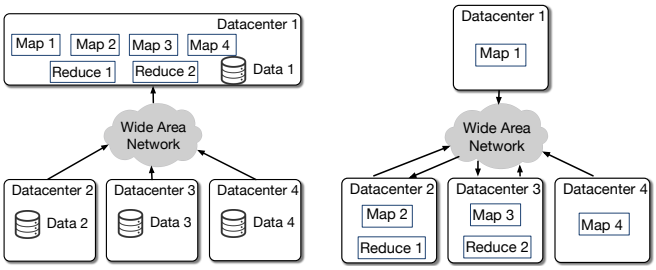
***Abstract*—Typically called big data processing, processing large volumes of data from geographically distributed regions with machine learning algorithms has emerged as an important analytical tool for governments and multinational corporations. The traditional wisdom calls for the collection of all the data across the world to a central datacenter 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 where the data is, while data should be processed with a minimum amount of transfers across datacenters. In this paper, we design and implement *Flutter*, a new task scheduling algorithm that improves the completion times of big data processing jobs across geographically distributed datacenters. To cater to the specific characteristics of data-parallel applications, we first formulate our problem as a lexicographical min-max integer linear programming (ILP) problem, and then transform it into a nonlinear program with a separable convex objective function and a totally unimodular constraint matrix, which can be solved using a standard linear programming solver efficiently in an online fashion. Our implementation of *Flutter* is based on Apache Spark, a modern framework popular for big data processing. Our experimental results have shown that we can reduce the job completion time by up to 25%, and the amount of traffic transferred among datacenters by up to 75%.**

I. Introduction

It has now become commonly accepted that the volume of data — from end users, sensors, and algorithms alike — has been growing exponentially, and mostly stored in geographically distributed datacenters 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 day, big data processing has become a routine in governments and multinational corporations, especially those in the business of social media and Internet advertising.

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 datacenter, so that they can be processed in a centralized fashion. 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 [1]. Second, the cost, in terms of both bandwidth and time, to move large volumes of data across geo-distributed datacenters may become prohibitive as the volume of data grows exponentially.

*This work is supported in part by the SAVI NSERC Strategic Networks grant.



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

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

It has been pointed out that [1], [2], [3], rather than transferring data across datacenters, 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 datacenter. Of course, the intermediate results after such processing may still need to be transferred across datacenters, 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 datacenters is shown in Fig. 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 datacenters. Yet, previous works (*e.g.,* [3]) were designed with assumptions that were often unrealistic — such as bottlenecks do not occur on inter-datacenter 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 complex prediction algorithms. Even if such knowledge were available, a big 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 [4].

In this paper, we have designed and implemented *Flutter,* a new system to schedule reduce tasks in Spark across data-

centers over the wide area. 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 designed to be *stage-aware*: it minimizes the completion time of each stage in a job, which corresponds to the slowest of the completion times of the constituent tasks in the stage. A stage in Spark means a group of independent tasks executing the same functions and having the same shuffle dependencies [5].

Practicality also implies that our algorithm in *Flutter* would need to be efficient at runtime. Our problem of stage-aware online scheduling can be formulated as a lexicographical min-max integer linear programming (ILP) problem. A highlight of this paper is that, 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.

To demonstrate that it is amenable to practical implementations, we have implemented *Flutter* based on Apache Spark, a modern framework popular for big data processing. Our experimental results on a production wide-area network with geo-distributed servers have shown that we can reduce the job completion time by up to 25%, and the amount of traffic transferred among different datacenters by up to 75%.

II. Flutter: Motivation and Problem Formulation

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 4 cores and 15 GB of main memory each. To illustrate the actual available capacities on inter-datacenter links, we have measured the bandwidth available across datacenters using the `iperf` utility, and our results are shown in Table I.

From this table, we can make two observations with convincing evidence. On one hand, when VMs in the same datacenter communicate with each other across the intra-datacenter network, the available bandwidth is consistently high, at around 1 Gbps. This is sufficient for typical Spark-based data-parallel applications [6]. On the other hand, bandwidth across datacenters is an order of magnitude lower, and varies significantly for different inter-datacenter links. For example, the link with the highest bandwidth is 175 Mbps, while the lowest is only 49 Mbps.

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

TABLE I
Available bandwidths across geo-distributed datacenters.

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

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 [5] 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 length of e represents the transfer time of intermediate results from the source node to the destination node of e.

Scheduling all the tasks in the DAG to a number of worker nodes — while minimizing the completion time of the job — is known as a NP-Complete problem in general [4], 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 practical way of designing a task scheduler, *Flutter* follows suit and only schedules the tasks within the same stage to geo-distributed datacenters, rather than considering all the ready tasks in the DAG. Here we denote the set of tasks in a stage by $\mathcal{N} = \{1 \dots n\}$, and the set of datacenters by $\mathcal{D} = \{1 \dots d\}$.

There is, however, one more complication when tasks within the same stage are to be scheduled. The complication 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 datacenter with a much longer transfer time to receive all the intermediate results needed (due to capacity limitations on inter-datacenter 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 as follows:

$$\operatorname*{lexmin}_{X} \quad \max_{i,j}(x_{ij} \cdot (c_{ij} + e_{ij})) \tag{1}$$

$$\text{s.t.} \quad \sum_{j=1}^{d} x_{ij} = 1, \qquad \forall i \in \mathcal{N} \tag{2}$$

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

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

$$x_{ij} \in \{0, 1\}, \qquad \forall i \in \mathcal{N},\ \forall j \in \mathcal{D} \tag{5}$$

In our objective function (1), $x_{ij} = 1$ indicates the assignment of the i-th task to j-th datacenter; otherwise $x_{ij} = 0$. c_{ij} is the transfer time to receive all the intermediate results, computed in Eq. (4). e_{ij} denotes the execution time of the i-th task in the j-th datacenter. Our objective is to minimize the maximum task completion time within a stage, including both the network transfer time and the task execution time.

To achieve this objective, there are four constraints that we will need to satisfy. The first constraint in Eq. (2) implies that each task should be scheduled to only one datacenter. The second constraint, Eq. (3), implies that the number of tasks assigned to the j-th datacenter should not exceed the maximum number of tasks f_j that can be scheduled on the existing VMs on that datacenter. Though it is indeed conceivable to launch new VMs on-demand, it takes a few minutes in reality to initiate and launch a new VM, making it far from practical. The total number of tasks that can be scheduled depends on the number of VMs that have already been initiated, which is limited due to budgetary constraints.

The third constraint, Eq. (4), is to compute the transfer time of the i-th task on j-th datacenter, where s_i and d_k represent the number of inputs for the i-th task and the index of the datacenter that has the k-th input, respectively. For example, let $m_{d_k j}$ denote the amount of bytes that need to be transfered from the d_k-th datacenter to the j-th datacenter if the i-th task is scheduled to the j-th datacenter. If $d_k = j$, then $m_{d_k j} = 0$. We let b_{uv} to denote the bandwidth between the u-th datacenter and the v-th datacenter, and assume that the network bandwidths $B_{d\times d} = \{b_{uv} |\ u,\ v = 1 \dots d\}$ across all the datacenters can be measured, and are stable over a few minutes. We can then compute the maximum transfer times for each possible way of scheduling the i-th task. The last constraint indicates that x_{ij} is a binary variable.

III. Network-aware Task Scheduling across Geo-Distributed Datacenters

Given the formal problem formulation of our network-aware task scheduling across geo-distributed datacenters, we now study how we solve the proposed ILP problem efficiently, which is the key for the practicality of *Flutter* in the real data processing systems. In this section, we first propose to transform the lexicographical min-max integer problem in our original formulation into a special class of nonlinear programming problem. We then further transform this special class of nonlinear programming problem into a *linear programming problem* (LP) that can be solved efficiently with standard linear programming solvers.

A. Transform into a Nonlinear Programming Problem

The special class of nonlinear programs that can be transformed into a LP should meet two conditions [7], a separable convex objective function and a totally unimodular constraint matrix. We will show how we transform our original formulation to meet these two conditions.

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 is to find a vector that is lexicographically minimal over all the feasible vectors with its components rearranged in a non-increasing order. In our problem, if $\mathbf{p}$ is lexicographically no greater than $\mathbf{q}$, then the 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 find out that we can use a summation of exponents to preserve the lexicographical order among vectors. Consider the following convex function $g : \mathbb{Z}^k \to R$ that has the form of

$$g(\boldsymbol{\lambda}) = \sum_{i=1}^{k} k^{\lambda_i},$$

where $\boldsymbol{\lambda} = \{\lambda_i \mid i = 1 \dots k\}$ $(k > 1)$ 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[1].

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

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 $\overrightarrow{q}_r \geq \overrightarrow{p}_r + 1$. Then we have:

$$\begin{aligned}
g(\mathbf{q}) - g(\mathbf{p}) = &\quad g(\overrightarrow{\mathbf{q}}) - g(\overrightarrow{\mathbf{p}}) && (6)\\
= &\quad \sum_{i=1}^{k} k^{\overrightarrow{q}_i} - \sum_{i=1}^{k} k^{\overrightarrow{p}_i} && (7)\\
= &\quad \sum_{i=r}^{k} k^{\overrightarrow{q}_i} - \sum_{i=r}^{k} k^{\overrightarrow{p}_i} && (8)\\
> &\quad \sum_{i=r}^{k} k^{\overrightarrow{q}_i} - k \times k^{\overrightarrow{p}_i} && (9)\\
= &\quad (k^{\overrightarrow{q}_r} - k^{\overrightarrow{p}_r + 1}) + \sum_{i=r+1}^{k} k^{\overrightarrow{q}_i} && (10)\\
> &\quad 0 && (11)
\end{aligned}$$

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 $\overrightarrow{p}_i = \overrightarrow{q}_i$

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

for all $i < r$.

$$g(\mathbf{q}) - g(\mathbf{p}) = g(\overrightarrow{\mathbf{q}}) - g(\overrightarrow{\mathbf{p}}) \quad (12)$$

$$= \sum_{i=1}^{k} k^{\overrightarrow{q}_i} - \sum_{i=1}^{k} k^{\overrightarrow{p}_i} \quad (13)$$

$$< \sum_{i=1}^{r-1} k^{\overrightarrow{q}_i} + (k+1-r) \times k^{\overrightarrow{q}_r} \quad (14)$$

$$- \sum_{i=1}^{r-1} k^{\overrightarrow{p}_i} - k^{\overrightarrow{p}_r} \quad (15)$$

$$= (k+1-r) \times k^{\overrightarrow{q}_r} - k^{\overrightarrow{p}_r} \quad (16)$$

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 to 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. (1). 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 also

$$\min \quad \sum_{i}^{n} \sum_{j}^{d} k^{x_{ij} \cdot (c_{ij} + e_{ij})}, \quad (17)$$

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. (17) 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 form a totally unimodular matrix.

2) Totally Unimodular Constraint Matrix: A totally unimodular matrix is an important concept because it can quickly determine whether a 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 if we can prove that the coefficients in the constraints of our formulation form a totally unimodular matrix, then our problem would only have integral solutions. We prove that the coefficients of the constraints in our problem formulation form a totally unimodular matrix by the following lemma.

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

Proof: A totally unimodular matrix is a $m \times r$ matrix $A = \{a_{ij} \mid i = 1 \ldots m,\ j = 1 \ldots r\}$ 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 $\mathcal{I} \subseteq \{1 \ldots m\}$, 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 $X = \{x_{ij} \mid i = 1 \ldots n,\ j = 1 \ldots d\}$ as a $nd \times 1$ vector, then we can write down the constraint matrix in (2) 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 (2) and (3), we can always assign the rows related to (2) 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. ■

B. Transform the Nonlinear Programming Problem into a LP

We have transformed our integer programming problem into a nonlinear programming problem 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 a LP based on the method proposed in [7]. In this transformation, the optimal solutions would not change. The key transformation is named λ-representation as listed below.

$$f(x) = \sum_{h \in \mathcal{P}} f(h) \lambda_h \quad (18)$$

$$\sum_{h \in \mathcal{P}} h \lambda_h = x \quad (19)$$

$$\sum_{h \in \mathcal{P}} \lambda_h = 1 \quad (20)$$

$$\forall \lambda_h \in R^+, \forall h \in \mathcal{P} \quad (21)$$

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, which is a LP as listed below:

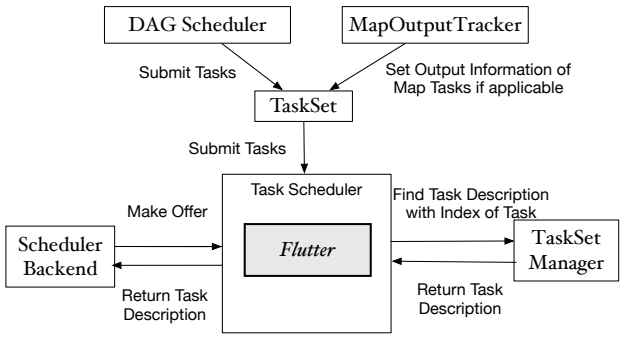


Fig. 2. The design of *Flutter* in Spark.

$$\min_{X,\boldsymbol{\lambda}} \quad \sum_{i=1}^{n}\sum_{j=1}^{d}\Big(\sum_{h\in\mathcal{P}} k^{(c_{ij}+e_{ij})\cdot h}\lambda_{ij}^{h}\Big) \tag{22}$$

$$\text{s.t.} \quad \sum_{h\in\mathcal{P}} h\lambda_{ij}^{h} = x_{ij},\ \forall i\in\mathcal{N},\ \forall j\in\mathcal{D} \tag{23}$$

$$\sum_{h\in\mathcal{P}} \lambda_{ij}^{h} = 1, \qquad \forall i\in\mathcal{N},\ \forall j\in\mathcal{D} \tag{24}$$

$$\lambda_{ij}^{h} \in R^{+}, \qquad \forall i\in\mathcal{N},\ \forall j\in\mathcal{D},\ \forall h\in\mathcal{P} \tag{25}$$

$$(2),(3),(4),(5). \tag{26}$$

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

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

$$\text{s.t.} \quad \lambda_{ij}^{1} = x_{ij},\ \forall i\in\mathcal{N},\ \forall j\in\mathcal{D} \tag{28}$$

$$(2),(3),(4),(5),(25). \tag{29}$$

We can clearly see that it is a LP with only nd variables, where n is the number of tasks and d is the number of datacenters. As it is a LP, it can be efficiently solved by standard linear programming solvers like Breeze [8] in Scala [9], and because the coefficients in the constraints form a totally unimodular matrix, its optimal solutions for X are integral and exactly the same as the solutions of the original ILP problem.

IV. Design and Implementation

After we 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 [10], Spark would cache 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 the 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* (RDD) and scheduling.

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 launched in Spark, the job would be transformed into DAG of tasks and then be handled by the DAG scheduler. After that, 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 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. 2. 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 obtain 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.

A. Obtaining Outputs of the Map Tasks

Flutter needs to compute the transfer time to obtain all the intermediate results for each reduce task if it is scheduled to one datacenter. 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.

B. Task Scheduling with Flutter

The task scheduler serves as a "bridge" that connects tasks and resources (executors in Spark). On 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 time of the tasks in the implementation because the execution times of the tasks in a stage are almost uniform. The second is to figure out the amount of available resources on each datacenter by analyzing received resource offers. After these two steps, we feed these information to our linear programming solver, and the solver would return an index of the most suitable datacenter for each reduce task. Finally, we randomly choose a host that has enough resource for the task on that datacenter and return the task description to the `SchedulerBackend` for launching the task. If the `TaskSet` does not have shuffle dependency, the default delay scheduling [11] 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.

V. Performance Evaluation

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

A. Experiemental Setup

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

Testbed: Our experiments are conducted on 6 datacenters with a total of 25 instances, among which two datacenters are in Toronto. The other datacenters are located at various academic institutions: Victoria, Carleton, Calgary and York. All the instances used in the experiments are m.large, which has 4 cores and 8 GB of main memory. The bandwidth capacities among VMs in these regions are measured by `iperf` and are shown in Table II.

The distributed file system used in our geo-distributed cluster is the Hadoop Distributed File System (HDFS) [10]. 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 [12] and GraphX [13]. Our scheduling algorithm is also applicable to Hadoop, while we choose Spark because our focus is on reduce tasks and there are multiple rounds of reduce tasks in iterative Spark jobs.

- **WordCount**: WordCount calculates the frequency of every single word appearing in a single file or a batch of files. It would first calculate the frequency of words in each partition, and then aggregate the results in the previous step to get the final result. We choose WordCount because it is a fundamental application in distributed data processing and it can be used to process the real-world data traces such as Wikipedia dump.
- **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 the ranks for the websites.
- **GraphX**: GraphX is a module built upon Spark for parallel graph processing. We run the application *LiveJournalPageRank* as the representative application 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 [14]. For GraphX, we adopt a directed graph in *LiveJournal* online social network with 4847571 nodes and 68993773 edges [14], where *LiveJournal* is a free online community.

Baseline: We compare our task scheduler with delay scheduling [11], which is the default task scheduler in Spark.

Metrics: The first two metrics used are job completion times and stage completion times of the three application. As the bandwidths among different datacenters are expensive in terms of cost, so we also take the amount of traffic transferred among different datacenters as another metric. Moreover, we also report the running times of solving the LP in different scales to show the scalability of our approach.

B. Experimental Results

In our experiments, we wish to answer the following questions. (1) What are the benefits of *Flutter* in terms of job completion times, stage completion times, as well as the volume of data transferred among different datacenters? (2) Is *Flutter* scalable in terms of the times to compute the scheduling results, especially for short-running tasks?

TABLE II
AVAILABLE BANDWIDTHS ACROSS GEO-DISTRIBUTED DATACENTERS (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 datacenters located at Toronto.

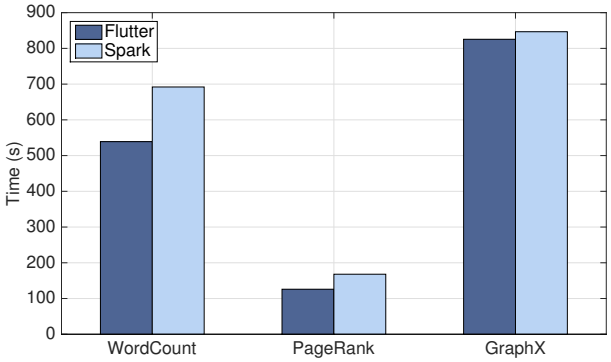


Fig. 3. The job computation times of the three workloads.

1) Job Completion Times: We plot the job completion times of the three applications in Fig. 3. As we can see that, completion times of all the 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 datacenter 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, thus it can significantly mitigate the stragglers — the slow-running tasks in that stage — and further improve the overall performance.

It seems that the improvements in terms of job completion times on GraphX are small, which may be because the total size of shuffle reads are relatively smaller than other applications, thus it only spends a small portion of the job completion time for shuffle reads, which limits the room for improvements. 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 datacenters for GraphX applications.

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. 4, we can thus 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 find out how the overall improvements of job completion times are achieved. We will explain the performance of the three applications one by one.

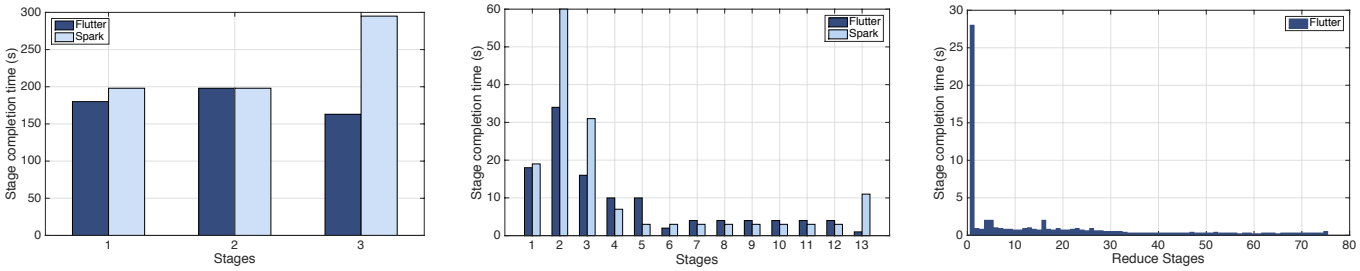
For WordCount, we repartition the input datasets as the input size is large. Therefore it has three stages as shown in Fig. 4(a). In the first stage, as it is not a stage with shuffle dependency, we use the default scheduler in Spark. Thus the performance achieved is almost the same. The second stage is a stage with shuffle dependency. We can see that the stage completion time of this stage for the two schedulers are almost the same, which is because the default scheduler also schedules the tasks in the same datacenters as ours while not necessarily in the same executors. In 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 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. 4(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 the first *distinct stage*, all the other stages are shuffle dependent. Thus we adopt *Flutter* instead of delay scheduling for task scheduling in those stages. As we can see that 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. 4(c) depicts the completion times of reduce stages in GraphX. As the total number of stages is more than 300, we only draw the stages named "reduce stage" 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. This may be for the reason that GraphX is designed for reducing the data movements and duplications, thus the stages can complete very quickly.

3) Data Volume Transferred across Datacenters: After we see the improvements of job completion times, we are now ready to evaluate the performance of *Flutter* in terms of the amount of data transferred across geo-distributed datacenters in Fig. 5. In WordCount, the amount of data transferred across different datacenters with the default scheduler is around three times to the one of *Flutter*. The amount of data across datacenters 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 datacenters is not the main goal of our optimization,



(a) WordCount (b) PageRank (c) GraphX

Fig. 4. The completion times of stages in WordCount, PageRank and GraphX.

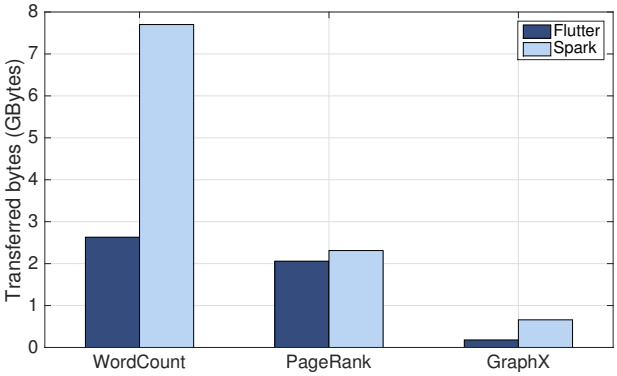


Fig. 5. The amount of data transferred among different datacenters.

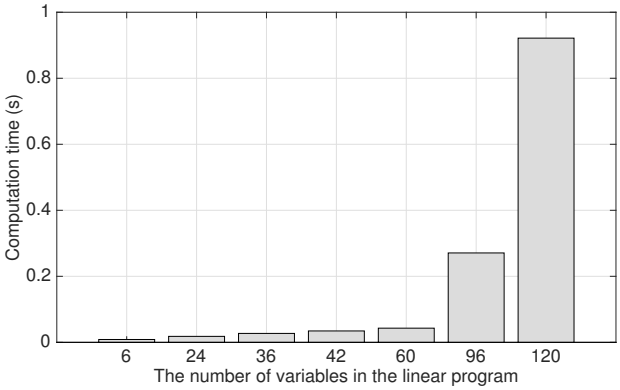


Fig. 6. The computation times of *Flutter*'s linear program at different scales.

we find out that it is in line with the goal of reducing the job completion time for data processing applications on distributed datacenters. This is because the bandwidth capacities across VMs in the same datacenter are higher than those on inter-datacenter links, so when *Flutter* tries to place the tasks to reduce the transfer times to get all the inputs, it may prefer to put the tasks in the datacenter that has most of the input data. Thus, it is able to reduce the volume of data transfers across different datacenters by a substantial margin.

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 when we run Spark applications. The results have been shown in Fig. 6. 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 a results 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 datacenters. *Flutter* is scalable for two reasons: (1) it is formulated as an efficient LP; and (2) the number of variables in our problem is small because the number of datacenters and reduce tasks are both small in practice.

VI. Related Work

In this section, we first show a few most related work in geo-distributed big data processing, which can be roughly divided into two categories based on their objectives: reducing the amount of traffic transferred among different datacenters and shortening the whole job completion time. We then survey some other work related to scheduling in distributed data processing systems.

Reducing the amount of traffic among different datacenters is proposed in [1], [2], [15]. In [1], they 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 datacenter has limitless storage, they aggressively cache the results of prior queries to reduce the data transfers of subsequent queries. In Pixida [2], 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 datacenter. However these solutions only address bandwidth cost without carefully considering the job completion time.

The most related recent work is Iridium [3] for low latency geo-distributed analysis, while we have some significant differences with it. First, they assume the network connecting the *sites* (datacenters) 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 bandwidth of VMs are both 1Gbps in intra datacenters, while the bandwidth among VMs in different datacenters are

only around 100Mbps. Therefore the network bottlenecks are more likely to exist in the network connecting the datacenters 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 not realistic assumptions as reduce tasks are not divisible with low overhead and the data skews are common in the data analysis frameworks [16]. While we use the exact amount of intermediate results that each reduce task would read from the outputs of map tasks. What is more, although they formulate the scheduling problem as a LP, in their implementation, they actually schedule the tasks by solving a mixed integer programming (MIP) problem as stated in their paper [3]. Besides Iridium, G-MR [17] is about executing a sequence of MapReduce jobs on geo-distributed data sets with improved performance in terms of both job completion time and cost.

For scheduling in data processing systems, Yarn [18] and Mesos [19] are the cluster managers designed for improving cluster utilization. Sparrow [20] is a decentralized scheduling system for Spark that can schedule a great amount of jobs at the same time with small scheduling delays, and Hopper [21] is an unified speculation-aware scheduling framework for both centralized and decentralized schedulers. Quincy [22] is designed for scheduling tasks with both locality and fairness constraints. Moreover, there is plenty of work related to data locality such as [11], [23], [24], [25].

VII. Concluding Remarks

In this paper, we focus on how tasks may be scheduled closer to the data across geo-distributed datacenters. We first find out that the network could be a bottleneck for geo-distributed big data processing, by measuring available bandwidths across Amazon EC2 datacenters. Our problem is then formulated as an integer linear programming problem, considering both the network and the computational resource constraints. To achieve both optimal results and high efficiency of the scheduling process, we are able to transform the integer linear programming problem into a linear programming problem, with exactly the same optimal solutions.

Based on these theoretical insights, we have designed and implemented *Flutter*, a new framework for scheduling tasks across geo-distributed datacenters. With real-world performance evaluation using an inter-datacenter network testbed, 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 across different datacenters. As part of our future work, we will investigate how data placement, replication strategies, and task scheduling can be jointly optimized for even better performance in the context of wide-area big data processing.

References

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

[2] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Technical Report: Pixida: Optimizing Data Parallel Jobs in Bandwidth-Skewed Environments," Tech. Rep., 2015.

[3] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, V. Bahl, and I. Stoica, "Low Latency Analytics of Geo-distributed Data in the Wide Area," in *Proc. of ACM SIGCOMM*, 2015.

[4] 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.

[5] 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. of USENIX NSDI*, 2012.

[6] 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, pp. 293–307.

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

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

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

[11] 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, pp. 265–278.

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

[13] 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. of USENIX OSDI*, 2014, pp. 599–613.

[14] 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.

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

[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proc. of ACM SIGMOD*, 2012, pp. 25–36.

[17] C. Jayalath, J. Stephen, and P. Eugster, "From the Cloud to the Atmosphere: Running MapReduce Across Data Centers," *IEEE Transactions on Computers*, vol. 63, no. 1, pp. 74–87, 2014.

[18] 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. of ACM SoCC*, 2013.

[19] 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. of USENIX NSDI*, 2011.

[20] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proc. of ACM SOSP*, 2013, pp. 69–84.

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

[22] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *Proc. of the ACM SIGOPS*, 2009, pp. 261–276.

[23] X. Zhang, Z. Zhong, S. Feng, B. Tu, and J. Fan, "Improving Data Locality of MapReduce by Scheduling in Homogeneous Computing Environments," in *IEEE 9th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2011, pp. 120–126.

[24] M. Hammoud and M. F. Sakr, "Locality-aware Reduce Task Scheduling for MapReduce," in *IEEE Conference on Cloud Computing Technology and Science (CloudCom)*, 2011, pp. 570–576.

[25] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving Mapreduce Performance through Data Placement in Heterogeneous Hadoop Clusters," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 2010, pp. 1–9.