When Network Matters: Data Center Scheduling with Network Tasks

Frédéric Giroire, Nicolas Huin, Andrea Tomassilli, Stéphane Pérennes

When Network Matters: Data Center Scheduling with Network Tasks

F. Giroire*, N. Huin†, A. Tomassilli*, and S. Pérennes*
*Université Côte d'Azur, CNRS, Inria Sophia Antipolis, France
†Concordia University, Montreal (Qc) Canada

***Abstract*—We consider the placement of jobs inside a data center. Traditionally, this is done by a task orchestrator without taking into account network constraints. According to recent studies, network transfers represent up to 50% of the completion time of classical jobs. Thus, network resources must be considered when placing jobs in a data center.**

In this paper, we propose a new scheduling framework, introducing network tasks that need to be executed on network machines alongside traditional (CPU) tasks. The model takes into account the competition between communications for the network resources, which is not considered in the formerly proposed scheduling models with communication. Network transfers inside a data center can be easily modeled in our framework. As we show, classical algorithms do not efficiently handle a limited amount of network bandwidth. We thus propose new provably efficient algorithms with the goal of minimizing the makespan in this framework. We show their efficiency and the importance of taking into consideration network capacity through extensive simulations on workflows built from Google data center traces.

I. Introduction

The increasing need for efficiently processing and analyzing huge amounts of data has led to data-oriented parallel computing solutions such as MapReduce [1], Dryad [2], CIEL [3], and Spark [4]. These solutions are based on input data partitioning over a number of parallel machines. Jobs are split up into finer–grained tasks, and partial results from the various stages of computation are then transferred through the network. Each stage requires all the outputs of the previous stage to be in place before moving to the next stage.

In this context, the network starts to become an increasingly significant bottleneck in the performance of parallel processing [5], [6] and hence, an important resource to optimize in a data center. Indeed, decreasing the parallel communications' completion time may lead to completing the corresponding job faster [7], [8], [9].

Today's most common applications spend a significant portion of their time in communications. As reported by [7], the communications accounted for 33% of total completion times of MapReduce jobs in Facebook's Hadoop cluster, and 42% for Monarch [10], an iterative MapReduce application in Spark identifying spam links on Twitter. The recent development of containers and micro-services [11] will amplify this trend. They further divide monolithic tasks into several subtasks, increasing the number of communications in the network.

Usually, when a job arrives, the orchestrator tries to optimize the data center resources and decide on which servers the job's tasks should be executed. Traditionally, this is done using scheduling algorithms which take into account properties of the server, such as CPU usage and memory utilization, and of the task, such as execution time, task deadline, and task activation time. The effects of the placement of the tasks on the network's resources are not usually taken into consideration. However, taking into account network resources when placing tasks is now of primary importance for a large number of applications to reduce the communication overhead.

Some scheduling models have been introduced to this end, such as *Scheduling with communication delays*, or *with communications costs*. If on one hand, they take into account communication delays, on the other hand, they do not consider the fact that network bandwidth might be limited and that the communications may compete for it, leading to an additional delay or cost when a large number of communications are performed at the same time.

We thus introduce a *new scheduling framework* which takes into account the limited communication bandwidth. In this framework, traditional (CPU) tasks stand alongside new *network tasks*. As usual, (CPU) tasks have to be executed by servers, but network tasks have to be executed by *network machines*, aiming to model the limited network capacity. The originality and difficulty of this study, compared to scheduling with non-identical machines, lies in the fact that *network tasks may or may not be executed depending on the placement of the CPU tasks*.

Indeed, when placing two CPU tasks T_1 and T_2, we would incur a communication delay only in the case in which T_1 and T_2 are scheduled on two different CPU machines. In such a case, we would have a network task $T_{1\to 2}$ to schedule on one of the network machines.

Our contributions can be summarized as follows.

- We introduce a new scheduling framework to model communication delays when tasks are competing for a limited network bandwidth. The idea is to model communications with network tasks which have to be executed on network machines.
- We show that the problem of scheduling data center jobs while routing their communications can be modeled with our scheduling framework using a simple set of network machines.
- We then study a new problem, Scheduling With Network Tasks, with the goal of minimizing the makespan of a set of tasks. The problem is NP-complete and we show that the simple 3-approximation List Scheduling algorithm with communication delays may be as bad as the simple algorithm putting all tasks on a single machine when the

network bandwidth is limited.

- We then propose a generalized version of the classical List Scheduling algorithm for our framework, called GENERALIZED LIST SCHEDULING. We show that our algorithm is optimal on simple MapReduce workflows.

- We also introduce a two-phase algorithm, PARTITION. In the first phase, the algorithm partitions the tasks into the available machines. In the second phase, we schedule at which time and in which order the tasks should be done. We provide approximation algorithms for the two phases.

- Finally, we evaluate the practical behavior of our algorithms. We perform extensive simulations using workflows based on Google Trace statistics [12]. We show that our algorithms are very efficient for scenarios in which network capacity has to be taken into account.

The rest of this paper is organized as follows. In Section II, we review related works in more detail. We then formally introduce the new framework and scheduling problem formally in Section III. We discuss the hardness of the problem in Section IV. We then propose two algorithms in Section V and we evaluate them in Section VI. Finally, we draw our conclusions in Section VII.

II. RELATED WORK

Optimizing Data Center Communications. Recent works have started to address the problem of optimizing network activity in order to improve job performance.

In [7], the authors propose a centralized application-level mechanism to coordinate transfers in the shuffle stage of MapReduce jobs with the goal of reducing the average job completion time. Indeed, according to their measurements on MapReduce applications, up to 50% of the completion time may be consumed in the shuffle phase. To this end, the authors propose a method based on weighted fair sharing at the cluster level. They show that, with their approach, the shuffle phase duration can be reduced by a factor of 1.5.

A family of works is based on the coflow abstraction [13], that is, a collection of parallel flows belonging to the same job. Varys [8] is a coordinated coflow scheduler designed with the goal of maintaining high network utilization and guaranteeing starvation freedom. [7] and [8] are centralized coflow schedulers. A decentralized solution is presented in [9]. The authors design and implement Baraat, a decentralized task-aware scheduling system for data centers. The goal is to minimize task completion time. Their solution is based on scheduling network resources at the unit of a task. They show that FIFO–based schemes perform well, allowing to reduce both the average and the tail task completion times.

In [14], the authors consider the problem of scheduling all three phases (i.e., Map, Shuffle, and Reduce) of the MapReduce process. They develop constant factor approximation algorithms to minimize the mean response time over all jobs.

Corral [15] is an offline planning algorithm with the goal of jointly optimizing the placement of data and compute, and improving the application latency. Corral performs network-aware task placement. Large shuffles are separated from each other to reduce network contention in the cluster and jobs are run across a small number of racks to their data locality.

In this paper, we introduce a *new theoretical framework* to address the problems considered in these works. In this framework, we define a new problem, SCHEDULING WITH NETWORK TASKS, and we propose (provably) efficient algorithms to solve it.

Scheduling. The problem of the paper SCHEDULING WITH NETWORK TASKS is related to different classic scheduling problems, see e.g., [16] for a comprehensive survey.

Scheduling with *precedence constraints* or *list scheduling* was introduced in [17]. The authors model the precedence with a directed acyclic graph (DAG), in which an arc D_{ij} between two tasks T_i and T_j means that task T_i has to be completely processed before T_j may begin its execution. The authors provide a $(2-1/m)$-approx of the problem. In the 90s, communications were introduced in the family of problems called *scheduling with communication delays.* At the end of a task, some data may be sent to other machines. A communication delay d_{ij} is paid to send data from machine i to machine j. The general problem of minimizing the makespan is still open (even with an infinite number of machines). However, when the communication delays are all the same for a given source ($d_i = d_{ij}$) and when a task can be duplicated on several machines to avoid some communication costs, there exists a 2-approximation algorithm [18]. When the communication costs are further simplified to be unitary, a 2-approximation exists without the additional hypothesis above [19].

In all the above models, no network capacity is assumed. The model of this paper, on the contrary, *takes into account the competition between flows to access a limited amount of network bandwidth.*

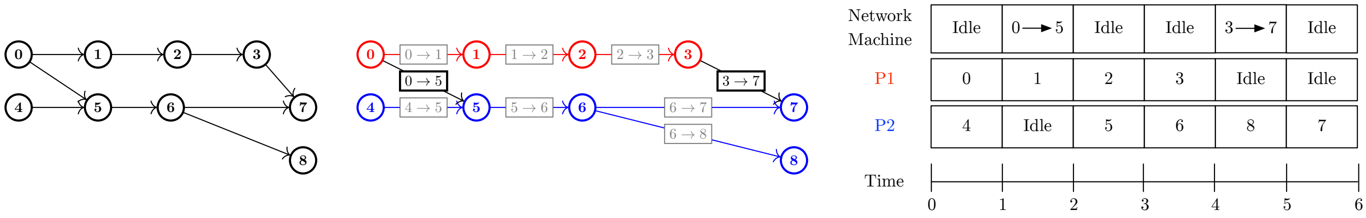
III. A NEW SCHEDULING FRAMEWORK

A. Problem and Example

We consider a set of jobs (often referred to as *workflows*) which have to be executed on m machines (also called processors or servers in the literature). A machine M_j has a processing speed S_j. Each job is made up of one or several tasks with dependency constraints between them. We denote by $\mathcal{T}$ the set of all the tasks (of all the jobs). The size of a task T_i is denoted by s_i. The completion time of a task T_i on the machine M_j is $c_{i,j} = s_i/S_j$.

The dependency between tasks in a job is expressed through a directed acyclic graph (DAG) in which an arc between tasks T_i and T_j means that T_i has to be completed before T_j may start. The set of jobs is thus modeled by a forest of DAGs.

When the task T_i has completed its execution, it may have computed data which is needed to execute task T_j. We model this by introducing a *network task* $T_{i\to j}$ which has to be executed after task T_i and before task T_j. The size of $T_{i\to j}$ is denoted by $s_{i\to j}$. Network tasks have to be executed on a set of k network machines, which represent network links (or communication channels). The network machine N_ℓ has a capacity C_ℓ. The completion time of a network task $T_{i\to j}$ on the network machine N_ℓ is $c_{i\to j,\ell} = s_{i\to j}/C_\ell$.



Network Machine	Idle	0 → 5	Idle	Idle	3 → 7	Idle
P1	0	1	2	3	Idle	Idle
P2	4	Idle	5	6	8	7

Fig. 1: (Left) The dependency (di)graph of a job J with 9 (CPU) tasks. (Middle) Modeling with network tasks indicated with rectangles. We provide a feasible schedule of job J. (Middle) Scheduled graph: (CPU) tasks executed by Machine 1 are in red, by Machine 2 in blue, carried out network tasks in black, not executed ones in gray. (Right) Timeline.

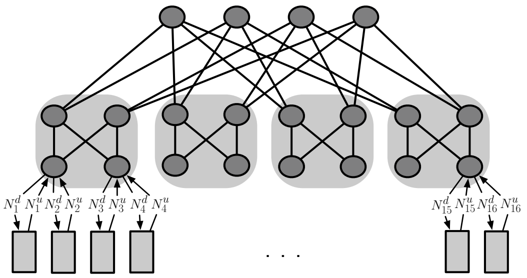

N_1^d N_1^u N_2^d N_2^u N_3^d N_3^u N_4^d N_4^u
. . .
N_{15}^d N_{15}^u N_{16}^d N_{16}^u

Fig. 2: Modeling data center communications with Network machines for a 4-Fat Tree with 16 (racks of) servers.

We now define the new scheduling problem.

Problem 1 (SCHEDULING WITH NETWORK TASKS). *Given a set of jobs $\mathcal{J}$ composed of tasks linked by a dependency digraph G and a set of network tasks $\mathcal{N}$, a set of m CPU machines, and k network machines, find the scheduling of $\mathcal{J}$ minimizing the makespan, that is, the maximum completion time of the jobs.*

Example: Consider a system with 2 machines, M_1, M_2, of processing speeds $S_1 = S_2 = 1$, and one network machine N_1 of capacity $C_1 = 1$. We want to execute the job J with 9 tasks and the dependency digraph given in Fig. 1. We also provide the digraph with the potential network tasks to be executed. We set all task sizes to one in this example.

In Fig. 1, we provide a feasible schedule for job J. At time 0, tasks T_0 and T_4 are the only tasks which may be executed. They are placed on machines M_1 and M_2 respectively. Their completion time is 1 (size/processing speed). At time 2, T_1 can be executed on M_1, as its only predecessor, T_0, has been executed, and as its result is available in M_1. All predecessors of T_5 have been executed, but the task cannot be carried out, as the result of T_0 is in M_1 and the one of T_4 is in M_2. The result of T_0 is thus sent to M_2, i.e., the network task $T_{0\to5}$ is executed by the network machine. It takes one time unit (size/capacity). The job completion time is thus 6.

B. Modeling Data Center Orchestration with Communication

We show here that we can model data center task orchestration and network resource allocation using our scheduling framework with a simple set of network machines.

Preliminary. Our framework directly models simple networks such as a set of machines connected via a bus by using a model with a single network machine or connected via an antenna (WiFi, 4G, ...) using a model with one network machine per channel. However, more complex networks can be represented.

Data center networks. The *simplicity of the model lies in the fact that only border links have to be modeled.* Indeed, data center architectures are built with large bisection bandwidth [20]. Topologies such as Fat Trees or VL2 have full bisection bandwidth. In fact, they are permutation networks. It means that when the capacity is available at the border to send and receive a communication, it is always possible to find a path inside the network for the communication. Thus, only border links (i.e., links between the servers and the ToR switches) have to be taken into account.

We consider a data center with m servers, see an example in Fig. 2. In large data centers, what we refer to as servers are in fact a rack of servers. In this case, we only model inter-rack bandwidth, as within-rack bandwidth is usually 5 to 20 times larger than inter-rack bandwidth [21]. Each server is modeled by a machine. However, we now introduce two network machines per link connecting a Top of Rack (ToR) switch to a server M_j, one for the download traffic, N_j^d, and the other for the upload traffic, N_j^u. When a network task is scheduled to be executed at time t between machines M_i and M_j, the task $T_{i\to j}$ is placed in both the upload network machine N_i^u of M_i and the download network machine N_j^d of M_j in the same time step t. The parallel execution of the task in both machines models the communication between the two machines.

In the following, we assume that the machines and the network machines are identical. For the machines, this is a classical case considered in scheduling problems and is often true in practice in data centers. For the network machines, they are all representing links between servers and ToR switches and thus are often similar in data centers. Thus, $c_{ij} = c_i$ for all tasks $T_i \in \mathcal{T}$ and $c_{i\to j,\ell} = c_{i\to j}$ for all $T_{i\to j} \in \mathcal{N}$.

Also, less efficient networks can be modeled. In this case, it is enough to reduce the capacity of the network machines by a factor equal to $\frac{C}{O(m \log m)}$, with C the minimum multicut of the network, to ensure that paths exist, see e.g., [22]. The model then gives a $\frac{C}{O(m \log m)}$-approximation of the makespan.

IV. HARDNESS

We show here that SCHEDULING WITH NETWORK TASKS is harder than scheduling with communication delays. Both problems are clearly NP-complete as they are generalizations

Algorithm 1 Generalized UET List scheduler

```
$U = \mathcal{T}$                          ▷ U is the set of unprocessed tasks
$t = 0$                                              ▷ $t$ is the clock
while $U \neq \emptyset$ do
    $t = t + 1$
    for $p = 1, 2, \ldots, m$ do                  ▷ Iterate on machines
        Compute the set of available tasks $A_{p,t}$
        if $A_{p,t} \neq \emptyset$ then
            min = $\{T' \in A_{p,t} | T' \sqsubset T$ for all $T \in A_{p,t}\}$
            Allocate to machine $M_p$ the task min at time slot $(t-1, t]$
```

of the problem of scheduling with precedence. However, there exists a simple greedy algorithm which has a 3-approximation factor when there are communication delays (but an infinite network capacity). We prove that this algorithm may be arbitrarily bad in our framework (by arbitrarily we mean $\Omega(m)$-approximate, i.e., the algorithm does not do significantly better than the simple algorithm putting all jobs on a single machine).

A. List-Scheduling

Next, we study the performance of the well-known "List Scheduling" algorithm which is 3-approximate in the case of infinite network capacity, i.e., b = $+\infty$ (see [19]). Initially, we describe the algorithm and then we show that its approximation ratio is bad in the worst case, even when considering only unit time tasks.

List scheduler. The UET list scheduler algorithm [19] provides a 2-approximation of the problem *scheduling with communication delays* when (CPU) task completion times and communication delays are unitary. We say that a task $T_j \in \mathcal{T}$ is available on Machine $M_p \in M$ during the time slot $(t-1, t]$ if it has no parent or if each of its parents has been completed either on machine M_i at time $< t-1$ or on machine $M_j \neq M_i$ at time $< t-2$. Note that, in this case, T_j can be feasibly executed by M_p during $(t-1, t]$.

Initially, the algorithm computes a total order $\sqsubset$ of the tasks of $\mathcal{T}$ (containing the partial order defined by the jobs) which corresponds to a feasible schedule if all tasks are executed on a single machine. Then, it produces a schedule by proceeding time slot by time slot and machine by machine deciding a subset of available tasks to be executed during the slot $(t-1, t]$. The pseudo-code of the algorithm is provided in Algorithm 1.

Efficiency when bandwidth is limited. The generalized UET List scheduler provides an ordered list of tasks to be executed for each machine. We now consider the same schedule in the scenario in which bandwidth is limited. In this case, a task which was scheduled at time t by the list scheduler may have to wait and be scheduled later after all the necessary communications are done. Note that the execution of the algorithm defines a natural (partial) order on the network tasks when executed with limited bandwidth. The network tasks of a task T_j with $T_i \sqsubset T_j$ cannot be executed before all network tasks of T_i have been executed. The partial order can then be extended arbitrarily to a total order.

Theorem 1. *List Scheduler is $\Omega(m)$-approximate when network bandwidth is limited even in the case of unitary costs.*

Proof. We consider an instance of the problem with m machines and one job with $m^2 + m + 1$ tasks, where the DAG of precedence constraints G consists of 3 layers of nodes. The first layer contains the tasks $T_1, T_2, \ldots, T_{m^2}$, the second layer consists of the tasks $T_{m^2+1}, T_{m^2+2}, \ldots, T_{m^2+m}$ while the third layer contains only the task T_{m^2+m+1}. Moreover, we have the following precedence relations: task T_{m^2+i} is preceded by tasks $T_{(i-1)m+1}, T_{(i-1)m+2}, \ldots, T_{im}$, for $1 \leq i \leq m$, and task T_{m^2+m+1} is preceded by tasks $T_{m^2+1}, T_{m^2+2}, \ldots, T_{m^2+m}$. There is a single ($k = 1$) network machine, N_1, of capacity $C_1 = 1$.

In the optimal schedule S^*, tasks $T_{(i-1)m+1}, T_{(i-1)m+2}, \ldots, T_{im}$ are executed by machine M_i followed by T_{m^2+i}, for $1 \leq i \leq m$. The task T_{m^2+m+1} is executed $m-1$ units of time after the completion of T_{m^2+1} on machine M_1. Only $m-1$ communications are performed during $(m+1, 2m]$ from $T_{m^2+2}, T_{m^2+3}, \ldots, T_{m^2+m}$ to T_{m^2+m+1}. The makespan of this schedule is $C(S) = 2m + 1$.

On the other hand, Algorithm 1 may choose an order scheduling tasks $T_{(i-1)m+1}, T_{(i-1)m+2}, \ldots, T_{im}$ simultaneously on machines $M_1, M_2, \ldots, M_m$, respectively, during the time slot $(i-1, i]$, for $1 \leq i \leq m$. The task T_{m^2+i} is executed by machine M_i. Lastly, the task T_{m^2+m+1} is executed by machine M_1. Globally, with Algorithm 1's schedule, $(m+1)(m-1)$ communications have to be done. $m(m-1)$ between tasks of Layers 1 and 2, and $(m-1)$ between tasks $T_{m^2+2}, ..., T_{m^2+m}$ and the task of Layer 3, T_{m^2+m+1}. No communication is done during the first and last time slot. During the other time slots, only one communication is performed. That is, the makespan computed by the algorithm is $C(S) = m^2 + 1$. □

The main problem of Algorithm 1's schedule is that it performs a large number of communications compared to the optimal solution. The trivial algorithm which schedules all tasks on a single machine and produces a schedule with no communications is obviously m-approximate and Theorem 1 implies that List Scheduling is at least as bad as this trivial algorithm. It is thus of primary interest to find efficient algorithms to deal with limited bandwidth.

V. Algorithms

In this section, we propose two algorithms to solve the problem of Scheduling With Network Tasks. The first one is a generalization of the well known List Scheduling algorithm. The second one divides the problem into two subproblems. The first subproblem computes an assignment of the tasks to machines while minimizing the CPU and the networking work. The second subproblem computes a schedule for the tasks once the placement has been selected. We provide approximation algorithms for the two subproblems.

A. Generalized List Scheduling

We propose a new algorithm, Generalized List Scheduling (referred to as G-List), to solve our problem.

Algorithm 2 GENERALIZED LIST SCHEDULING

```
$U = \mathcal{T}$                                ▷ U is the set of unprocessed tasks
$t = 0$                                          ▷ $t$ is the clock
while $U \neq \emptyset$ do
    $t = t + 1$
    compute $A$ the set of available task/machine-assignments.
    sort $A$ according to number of needed communications
    while $A \neq \emptyset$ do
        $(T_{\min}, M_{\min}) = \min(A)$
        Allocate to machine $M_{\min}$ the task $T_{\min}$ at time slot $(t-1, t]$
        Allocate needed network tasks for $T_{\min}$ to network machines in previous time slots
        Update $A$
```

To do so, we generalize the notion of an available task defined for the list scheduler algorithm [19]. The goal is then to avoid carrying out useless network tasks. The main idea is two-fold: (1) Like classical greedy algorithms, we consider tasks and their possible assignments to machines. However, the same task may need different amounts of communications if assigned to different machines. We thus consider all the possible (available task, machine) couples at time t and sort them according to the number of required communications by the schedule. The algorithm thus places a task on a machine in which the most data is available if possible. (2) A task is placed on a machine at time t only if all its communications tasks can be done before time t. Otherwise, we delay its placement.

Description of the algorithm. A high level pseudo-code of G-LIST is provided in Algorithm 2. We define an *available task/machine-assignment* at time slot $(t-1, t]$ as a pair task/machine $(T \in \mathcal{T}, M)$ for which all preceding tasks of T have been completed before time $t-1$ and for which all needed communications with machines different than P can be scheduled before time $t-1$. At each time slot, G-LIST computes the set A of available task/machine-assignments. It then sorts the tasks in the set according to the amount of needed communications to be scheduled. While A is not empty, it schedules $(T_{\min}, M_{\min})$, the minimum element of the set. It then updates A by removing the task/machine-assignments with machine $M_{\min}$ and the ones whose needed communications cannot be scheduled any more.

Note that A is not computed and sorted from scratch at each iteration. Indeed, A can be updated using simple efficient algorithms and data structures. Moreover, when completion times of tasks are large, it is not necessary to iterate on all time slots. Several features are added to improve the algorithm:

(1) In case of ties, we place first the task whose out-tree has the longest branch, considering the sum of CPU and network tasks. Indeed, the longest branch is a lower bound on the time to process the tasks depending on a task. It may be seen as a generalization of placing first tasks with longer processing times in the classic $4/3$-approximation algorithm for scheduling [17].

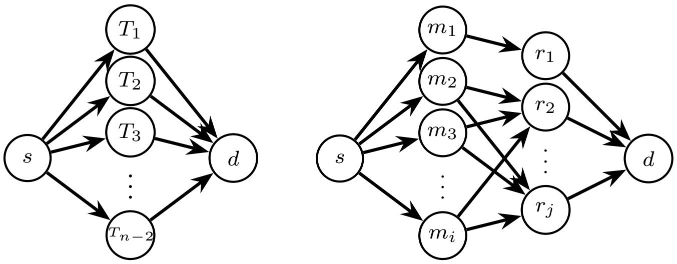


Fig. 3: Example of simple (left) and single-stage (right) MapReduce workflows with i map tasks and j reduce tasks.

(2) The algorithm makes two passes: the first one considering the workflow and the second one considering the "reverse workflow" in which an arc between task T_i and T_j is transformed into an arc between tasks T_j and T_i. Then, the best between the two passes is selected. The idea is that out-trees are optimized by the first pass and in-trees by the second pass, in the sense that tasks of the same subtrees are placed on the same machines if possible.

(3) For each job, we designate a longest branch as the master branch; all of its tasks are executed by a so called *master* machine. Then, before placing a task on a slave machine, we first test that it would not be faster to place it on the master machine when it will be free. That is, we only place a task on a slave machine if its completion time plus the time to send back the result to the master is smaller than the completion time of the master machine.

Discussion. Note that, when considering no dependency between tasks (and thus no communication), G-LIST boils down to the classical greedy scheduling algorithm which is a $4/3$ approximation. When considering dependency and no communication, it reduces to *list scheduling* of [17], and when considering dependencies and communication (but no bandwidth limit), to List Scheduler of [19].

B. Optimality on simple MapReduce Workflows

We consider the specific case of a simple MapReduce workflow, in which there is a single *Map* phase and a single *reduce* task. Formally, a simple MapReduce workflow is defined by a DAG with a source task s linked to $n-2$ tasks, $T_1, \ldots, T_{n-2}$, which are linked to a target task d, see Fig. 3. The completion times of tasks $T_1, \ldots, T_{n-2}$ are equal. Similarly, the communication times of tasks $T_{s \to T_i}$ for $1 \leq i \leq n-2$ are equal and the communication times of tasks $T_{T_i \to d}$ for $1 \leq i \leq n-2$ also. Note that simple MapReduce workflows are very frequent in data centers and that they also model simpler workflows (by setting to 0 some completion times) such as *Map* workflows and *Reduce* workflows which are also very common [23].

We prove here that G-LIST is optimal on a simple MapReduce Workflow. Note that the problem is NP-complete if the processing times of tasks $T_1, ..., T_{n-2}$ are different. Indeed, an instance of the k-PARTITION PROBLEM can be directly reduced to an instance of the problem, in which the numbers are the processing times of the tasks of a MapReduce workflow and for which the communication times are 0 and the capacity is infinite. The problem is NP-complete and no pseudo-polynomial algorithm exists to solve it when $k \geq 3$ [24].

Proposition 1. G-LIST *is optimal on simple MapReduce workflows.*

Proof. Let us compute the makespan of G-LIST on a simple MapReduce workflow. We note a (resp. b and c) the completion time of network tasks $T_{s\to T_i}$ (resp. of (CPU) task T_i and of network tasks $T_{T_i\to d}$) for $1 \leq i \leq n-1$.

G-LIST first selects any available branch (all branches are equivalent) of the workflow to be executed by a master machine. Note that without loss of generality we can always consider that the master machine executes both task s and d. The makespan is thus given by the time at which the master machine finishes the last job d, denoted by t_m.

We denote by κ, the number of intermediate tasks (out of the $n-2$) carried out by slave machines. The master thus carries out task s, $n-2-\kappa$ intermediate tasks, and task d.

$$t_m = c_s + \max((n-2-\kappa)b, t_s) + c_t,$$

where t_s is the time at which the slave machine receiving the last job (we refer to this machine as the last slave machine in the following) has finished sending its last result. Indeed, the task t is done when the master has finished all the $(n-2-\kappa)$ intermediate tasks assigned to it, after a time $(n-2-\kappa)b$, and once the last data was received after a time t_s given by

$$t_s = t_f + \left(\left\lceil \frac{\kappa}{m-1} \right\rceil - 1\right) t_I + b + t_\ell,$$

where t_f is the time at which the last slave machine receives its first data; the interjob time t_i is the time between the execution of two tasks; and t_ℓ is the time to send the result of the last job, starting from the end of its execution. We have $t_f = a(m-1)$ if $m-1$ divides κ, and $t_f = a(\kappa \mod (m-1))$ otherwise.

G-LIST sends a task to a slave machine only if it is faster to send its data, execute it, and get back its result, than executing it on the master machine. Thus, G-LIST selects

$$\kappa = \arg\min_{\kappa}(\max(n-2-\kappa)b, t_s)).$$

The last step is to show that the makespan of G-LIST is optimal. Due to lack of space, the proof is not provided here, but it can be found in [25]. □

C. PARTITION

When the workflows are complex, the greedy algorithm may have difficulty in assigning the tasks to the available machines while minimizing the network load. To prevent this, an efficient method consists in first carrying out a partition of the tasks to be done by machines while minimizing the network tasks that would be necessary to be done. We call this first phase or subproblem the PARTITIONING TO SCHEDULE. For this subproblem, we provide an approximation algorithm with factor $O(\sqrt{\log n \log m})$, with n being the number of tasks and m the number of machines, which comes from the best approximation factor for the $k-$balanced partitioning problem. When this preliminary phase is done, we just have to decide the order to process the tasks. We call this problem the SCHEDULING WHEN PLACED problem. We provide an algorithm (which is a generalization of Hu's algorithm to handle network tasks) which is a depth-approximation of the problem. Practical workflows have low depth, e.g., typically less than or equal to 4 for a MapReduce workflow. This leads to a constant factor approximation ratio in practice.

1) PARTITIONING TO SCHEDULE*:* To solve the problem, we use, as a subroutine, an algorithm to solve the classic k-balanced partitioning problem [26]. Given an integer $k \geq 2$ and a real $\nu \geq 1$, a (k,ν)-balanced partition of $G = (V,E)$ is a subset of the edges whose removal partitions the graph into at most k parts, where the sum of the vertex weights in each part is at most $\frac{\nu}{k} w(V)$. The (k,ν)-balanced partitioning problem with input $G = (V,E)$, k, and ν is to find a (k,ν)-balanced partition of G with minimum capacity, i.e., for which the sum of the weights of the arcs between parts is minimized. When $\nu = 2$, the problem is just called the k-balanced partitioning problem. Classic algorithms achieve an approximation factor of $O(\log n)$ to solve the problem [26], [27]. The approximation algorithm with the best known approximation factor, $O(\sqrt{\log n \log k})$, is due to Krauthgamer et al. [28].

PARTITION works as follows. We consider the undirected version of the DAG of the workflow as input. The completion times of the network (resp. CPU) tasks correspond to the weights of the edges (resp. of the vertices). As we do not know in advance if the best partition for our problem is balanced (indeed, if the network delays are very long, it may be better to schedule all the tasks on a single machine), we systematically test different levels of balance.

The algorithm solves the m k-balanced partitioning problems, for $1 \leq k \leq m$. It then outputs the best solution, that is, the one minimizing the sum of the weights of the arcs between parts divided by k (corresponding to the average work of the network machines) and the maximum partition size (corresponding to the work of the (CPU) machines).

In fact, first note that there exists an optimal partition using fewer than k machines when the maximum work over all machines is less or equal to $\frac{2n}{k} w(V)$. Indeed, if two machines have less than $\frac{n}{k} w(V)$ work to do, only one among both machines may do all the tasks assigned to them, and the makespan may not be increased. Thus, only one machine may have less than $\frac{n}{k} w(V)$ work to do, and there may be only $k-1$ machines with more.

Theorem 2. PARTITION-ASSIGN *provides a* $O(\sqrt{\log n \log m})$ *-approximation algorithm of the* PARTITIONING TO SCHEDULE *problem.*

Proof. Let $S^* = W^*_{CPU} + W^*_N$ be an optimal solution of the PARTITIONING TO SCHEDULE problem, where W^*_{CPU} is the maximum work to be done on a machine and W^*_N is the network work.

There exists an integer k, with $1 \leq k \leq m$, such that $\frac{n}{k} \leq W^*_{CPU} \leq \frac{2n}{k}$. Indeed, $W^*_{CPU} \geq \frac{n}{m}$ as at least one machine of the m machines has to do more than $1/m$-th of the work and $W^*_{CPU} \le n$, the total amount of work to be done.

Remark now that there exists an optimal partition using fewer than or exactly k machines when the maximum work over all machines is less than or equal to $\frac{2n}{k}w(V)$. Indeed, if two machines have less than $\frac{n}{k}w(V)$ work to do, only one among both machines may do all the tasks assigned to them, and the makespan may not be increased. Thus, only one machine may have less than $\frac{n}{k}w(V)$ work to do, and there may be only $k-1$ machines with more. Thus, without loss of generality, consider that S^* uses at most k machines.

Consider now the solution S_A provided by the k-balanced partitioning algorithm for this value of k. We have $S_A = \max\_part\_size +$ cut_weight, with cut_weight the capacity of its cut and $\max\_part\_size$ the maximum weight of a part.

On one hand, we have that cut_weight $\le O(\sqrt{\log n \log k})W^*_N$. Indeed, S^* provides a solution of the k-balanced partitioning algorithm, as it uses at most k machines. On the other hand, we have that $\max\_part\_size \le \frac{2n}{k}$. As $\frac{n}{k} \le W^*_{CPU}$, we get that $\max\_part\_size \le 2W^*_{CPU}$. This yields that $S_A \le O(\sqrt{\log n \log m})S^*$. □

Algorithm 3 PARTITION

```
Input: Set of workflows $G$, $m$ number of machines.
partitions[m]    ▷ Solutions of $m$ partitioning problems
for $k = 1, 2, \dots, m$ do ▷ Iterate on number of processors
    partitions[k] ← Compute an approximate solution of the $k$-balanced partitioning problem for $G$.
best_sol = $\min_k$(max_part_size($partitions[k]$) + cut_weight($partitions[k]$)$/k$)
return best_sol
```

As the algorithm of Krauthgamer et al. is based on semi–definite programming and has a long execution time, to solve the problem in practice we use the $O(\log n)$ approximation algorithm described in [27]. The main idea is to recursively partition the graph by solving, at each step, a Minimum Bisection Problem. We use the Kernighan and Lin heuristic algorithm [29] to solve bisection, leading to a time complexity of $O(mn^3 \log n)$.

2) SCHEDULING WHEN PLACED:

Theorem 3. PARTITION-SCHEDULE *provides a depth(W)-approximation algorithm of the* SCHEDULING WHEN PLACED *problem, where depth(W) is the depth of the workflow W to be scheduled.*

Proof. PARTITION-SCHEDULE considers the tasks of the workflow layer by layer. It does not schedule a task of layer j if a task of layer i with $i < j$ can be scheduled.

Consider an optimal schedule S^* and let $C(S^*)$ be its makespan. We denote by $C(L_i)$ the time to process the tasks of layer i and by $C(L_i \to L_j)$ the time to process the network tasks between layers i and j. Clearly, $C(S^*) \ge C(L_i)$ for $1 \le i \le depth(W)$. Similarly, $C(S^*) \ge C(L_i \to L_{i+1})$ for $1 \le i \le depth(W) - 1$. Thus, the makespan of PARTITION-SCHEDULE, $c(A)$, is such that

$$c(A) \le \sum_{i=1}^{depth(W)} C(L_i) + \sum_{i=1}^{depth(W)-1} C(L_i \to L_{i+1}).$$

We thus have $c(A) \le (2\ depth(W) - 1)C(S^*)$. □

VI. EXPERIMENTAL EVALUATION

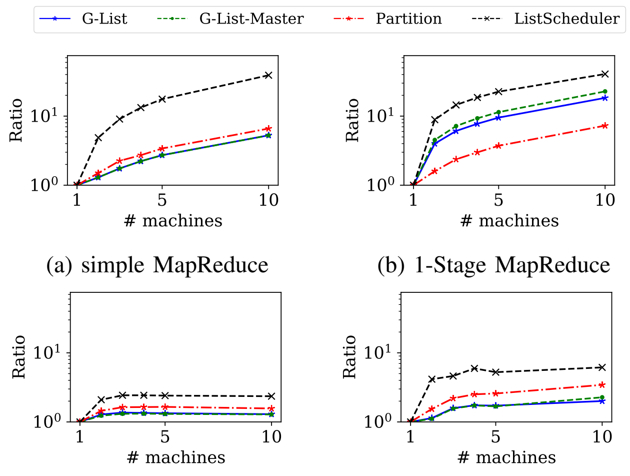
To validate our algorithm, we carried out some experiments using workflows built using statistics from the data center traces comprising 25 millions tasks released by Google [12]. We compare the performances of our two proposed algorithms, G-LIST (its variants with and without selection of the master branch referred to as G-LIST-MASTER and G-LIST, respectively) and PARTITION, with the ones of List Scheduler [19] which was proposed to handle communication delays, but which does not take into account the limited network capacity. We show the importance of taking into account the competition of tasks for bandwidth.

Trace. We extracted the distributions of the number of tasks per job and of the delay of computational tasks from the trace. The variances of the distributions are huge. Indeed, 75% of jobs have only 1 task, but these tasks only account for 20% of the total tasks. The average and maximum number of tasks of a workflow are 38 and 90,000, respectively. Also, the task completion time is heavy-tailed. The mean value is 28 minutes, but the longest task lasted 5 and a half days [30].

Network. The traces do not include statistics on network delays. This is why we tested different scenarios. To this end, we define the parameter ρ, which we refer to as *network factor* and which is the ratio between the average delay of a network task and the average delay of a CPU task. We then considered different values between 0% and 400% for ρ. We use $\rho = 0.5$ as the default value, as it corresponds to a scenario in which roughly 33% of the time is spent in network transfers [7].

Workflows. The dependencies between the tasks of a workflow are also not provided. We thus compare the algorithms using workflows of different types: simple MapReduce (defined in Section V-B), 1-Stage MapReduce, and Random workflows. 1-Stage MapReduce workflows contain a map phase, a shuffle phase, and a reduce phase (see Fig. 3). For a given number of tasks, we randomly choose the proportion of tasks in the first layer and in the second layer. We then connect task s to all the tasks of the first layer, and all the tasks of the second layer to task d. Each task of the first layer is then connected to a task in the second layer. We then choose the edge density of the workflow, that is, a probability p that a given task in the second layer is dependent of a given task of the first layer. Random workflows are built in the following way: we order the tasks from T_1 to T_n. To avoid cycles, we only add an arc from T_i to T_j (with probability p) if $i < j$.

Datasets. The datasets are then built in the following way. We choose a number of jobs to be executed. For each job, we choose its type of workflow randomly: simple MapReduce, 1-Stage MapReduce or Random workflow, with probabilities



(a) simple MapReduce

(b) 1-Stage MapReduce

(c) random workflow

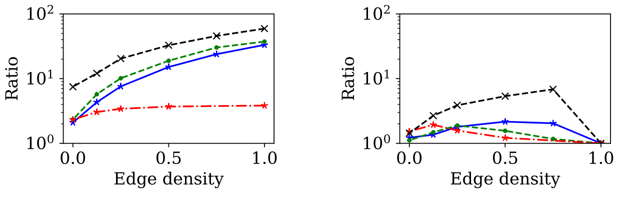
(d) 20 workflows

Fig. 4: Efficiency of the proposed algorithms as a function of the number of machines for different types of workflows.

20, 40, and 40%, respectively. It corresponds to a realistic distribution as at least 50% of applications in clusters can fit in the MapReduce paradigm [23]. We then draw its number of jobs randomly according to the distribution of the Google trace. The completion times of the (CPU) tasks (resp. of the network tasks) are then chosen according to the distribution of the Google trace (and multiplied by the network factor ρ). For each experiment, we average over 100 datasets.

Results. We compare the *makespan* of the schedules given by the different algorithms. We also study its sensitivity to different parameters such as the number of data center servers, number of tasks in a job, workflow edge density, and network factor. We provide two sets of results. The first ones are for a single workflow. The goal is to understand the efficiency of the algorithms for different types of workflows. The second ones are for sets of 20 random workflows of different types. Note that a single workflow may have up to 90,000 tasks. The goal is to see how efficient the algorithms are for a data center workflow. As the variance of the Google trace is very high, we present the results using a *normalized makespan metrics*, denoted as *ratio*. It is defined as the ratio between the makespan provided by an algorithm and the best of two classical lower bounds: $\sum_{T_i \in \mathcal{T}} c_i/m$ and the completion time of the longest branch. This metric allows to normalize the makespan between workflows with very different task completion times. It also gives an idea of the cost of network communications as the lower bound does not take into account the completion time of network tasks.

Number of machines. We first vary the number of machines used to execute the workflows (Fig. 4). With one machine, all the algorithms have a ratio of 1 as all the tasks are executed on 1 machine and no network tasks need to be done. When the number of machines increases, the ratio increases as tasks are done on several machines in order to decrease the makespan. For simple MapReduce workflows, the ratio increases to 5 even though G-LIST is optimal. It means that this value corresponds to the cost of network communications (and not to a gap to optimality). Note that, for other types of workflows (1-Stage MapReduce, random), the ratio has similar or lower values, showing the efficiency of our algorithms.



(a) 1-Stage MapReduce

(b) random workflows

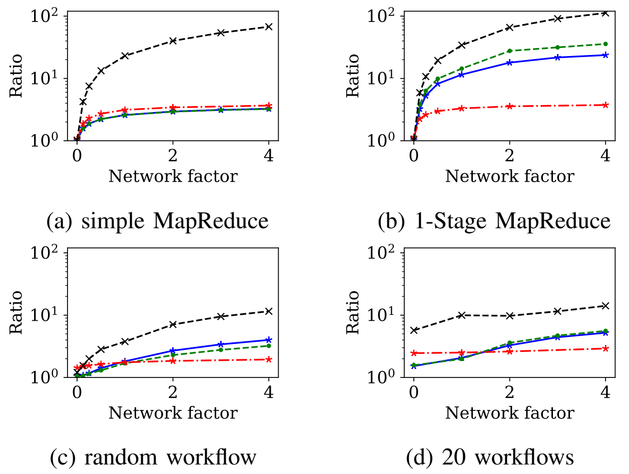
Fig. 5: Efficiency of the proposed algorithms as a function of the workflow edge density.

The gap between List Scheduler and our algorithms, G-LIST and PARTITION, also increases with the number of machines. This shows that our algorithms make much better use of the increased processing power available by optimizing the network communications. G-LIST provides the best solutions for simple MapReduce worfklows (Fig. 4a), as the algorithm is optimal for this type of workflow. However, PARTITION is also behaving well and provides close to optimal solutions. For 1-Stage MapReduce workflows (Fig. 4b), PARTITION is a lot more efficient than G-LIST as this type of workflow is more complex to schedule. On random workflows, both algorithms behave well. This is due to the fact that random workflows have DAGs with longer depths and that there are fewer possible scheduling combinations (see the following discussion for edge density). Indeed, the ratio is close to 1 in this case. On the sets of 20 workflows, the three algorithms behave similarly, with a small advantage for G-LIST-Master.

We also draw similar conclusions when varying the *number of tasks per workflow*. We did not include the corresponding plots due to the page limit.

Edge density. To understand for which kinds of workflows each algorithm is more efficient, we studied two parameters: edge density and network factor. We made the edge density vary from 0 to 1 (Fig. 5). With a small edge density, all algorithms behave well. The tasks are not very dependent on each other, and scheduling decisions are easy to take. When the edge density increases, scheduling becomes harder, and PARTITION behaves better as it considers the global structure of the dependency digraph, especially for 1-Stage MapReduce workflows (Fig. 5a). For random workflows (Fig. 5b), PARTITION is also better for edge densities higher than 0.2. However, all algorithms (including List Scheduler) behave well when the value of the edge density is 1. Indeed, in this case, there exists a complete order of the tasks, so all algorithms carry out the same schedule on a single machine. In general, random workflows tend to have a long branch. G-LIST-Master is executing all the tasks of this branch on the master machine and thus is the most effective for this type of workflow.

Network factor. We vary ρ from 0 to 4 (Fig. 6). When the network factor is zero, all algorithms are equivalent. Indeed,



(a) simple MapReduce

(b) 1-Stage MapReduce

(c) random workflow

(d) 20 workflows

Fig. 6: Efficiency of the proposed algorithms for different network factors and types of workflows.

this corresponds to a scenario in which network capacity is not a limiting resource. In this case, only the CPU task placement has to be optimized. Then, when the completion times of the network tasks increase, our algorithms, as expected, perform better than List Scheduler. PARTITION is the most efficient for all except for simple MapReduce workflows.

To summarize, both algorithms behave well for different types of workflows and different sets of parameters. G-LIST is the best on simple MapReduce worfklows and its variant with Master branch is efficient on Random workflows. PARTITION, in general, is better when the workflows are more complex and when the network is a strong bottleneck. Data center operators should thus choose a solution based on their mix of applications and network capacity.

VII. CONCLUSION

In this paper, we proposed a new framework to model the orchestration of tasks in a datacenter for scenarios in which the network bandwidth is a limiting resource. We introduce a new problem, SCHEDULING WITH NETWORK TASKS, in which, along with traditional (CPU) tasks, network tasks have to be scheduled on network machines. We propose two algorithms to solve the problem, G-LIST and PARTITION, for which we derive some theoretical guarantees. We demonstrate their effectiveness using datasets built using statistics from Google data center traces [12].
The paper focuses more on the theoretical side. An interesting future work may also concern the study of the practical behaviors of the algorithms on a testbed, comparing them with practical solutions proposed for data centers.

REFERENCES

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

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS operating systems review*, vol. 41, no. 3. ACM, 2007.

[3] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: a universal execution engine for distributed data-flow computing," in *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2011, pp. 113–126.

[4] 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 *USENIX conference on Networked Systems Design and Implementation*, 2012.

[5] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, vol. 39, no. 4, 2009, pp. 51–62.

[6] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, 2008.

[7] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, 2011.

[8] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, 2014, pp. 443–454.

[9] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *ACM SIGCOMM Computer Communication Review*, 2014, pp. 431–442.

[10] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, "Design and evaluation of a real-time url spam filtering service," in *IEEE Symposium on Security and Privacy (SP)*, 2011, pp. 447–462.

[11] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[12] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," *Google Inc., White Paper*, pp. 1–14, 2011.

[13] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *ACM HotNets*, 2012, pp. 31–36.

[14] F. Chen, M. Kodialam, and T. Lakshman, "Joint scheduling of processing and shuffle phases in mapreduce systems," in *IEEE INFOCOM 2012*.

[15] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 2015, pp. 407–420.

[16] B. Chen, C. N. Potts, and G. J. Woeginger, "A review of machine scheduling: Complexity, algorithms and approximability," in *Handbook of combinatorial optimization*. Springer, 1999, pp. 1493–1641.

[17] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[18] C. H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *SIAM journal on computing*, vol. 19, no. 2, pp. 322–328, 1990.

[19] V. J. Rayward-Smith, "Uet scheduling with unit interprocessor communication delays," *Discrete Applied Mathematics*, vol. 18, no. 1, 1987.

[20] T. Chen, X. Gao, and G. Chen, "The features, hardware, and architectures of data center networks: A survey," *Journal of Parallel and Distributed Computing*, vol. 96, pp. 45–74, 2016.

[21] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters." in *USENIX Annual Technical Conference*, 2014, pp. 1–12.

[22] N. Garg, V. V. Vazirani, and M. Yannakakis, "Approximate max-flow min-(multi) cut theorems and their applications," in *ACM symposium on Theory of computing*, 1993, pp. 698–707.

[23] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: an analysis of hadoop usage in scientific workloads," *Proceedings of the VLDB Endowment*, vol. 6, no. 10, pp. 853–864, 2013.

[24] R. G. Michael and S. J. David, "Computers and intractability: a guide to the theory of np-completeness," *WH Free. Co., San Fr*, 1979.

[25] F. Giroire, N. Huin, A. Tomassilli, and S. Pérennes, "When network matters: Data center scheduling with network tasks," Inria, Tech. Rep., Jan. 2019.

[26] G. Even, J. Naor, S. Rao, and B. Schieber, "Fast approximate graph partitioning algorithms," *SIAM Journal on Computing*, vol. 28, 1999.

[27] H. D. Simon and S.-H. Teng, "How good is recursive bisection?" *SIAM Journal on Scientific Computing*, vol. 18, no. 5, pp. 1436–1445, 1997.

[28] R. Krauthgamer, J. Naor, and R. Schwartz, "Partitioning graphs into balanced components," in *ACM-SIAM SODA 2009*.

[29] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, 1970.

[30] Z. Liu and S. Cho, "Characterizing machines and workloads on a google cluster," in *IEEE Parallel Processing Workshops (ICPPW)*, 2012.