Data Center Scheduling With Network Tasks

Frédéric Giroire, N. Huin, Andrea Tomassilli, S. Pérennes

Data Center Scheduling with Network Tasks

F. Giroire*, N. Huin†, A. Tomassilli*, and S. Pérennes*
* Université Côte d'Azur, CNRS, I3S, Inria, Sophia Antipolis, France
†IMT Atlantique, IRISA UMR CNRS 6074, F-35700 Rennes, France

***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 may account for 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 [2], Dryad [3], CIEL [4], and Spark [5]. 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 resources become an increasingly important bottleneck in the performance of parallel processing [6, 7] and hence, it is crucial to optimize them in a data center. In fact, reducing the completion time of parallel communications can lead to faster completion of the corresponding job [8, 9, 10].

Today's most common applications spend a significant portion of their time in communications. As reported by [8], the communications accounted for 33% of the total completion times of MapReduce jobs in Facebook's Hadoop cluster, and 42% for Monarch [11], an iterative MapReduce application in Spark identifying spam links on Twitter. This trend will become even more pronounced in the near future due to the recent development of containers and micro-services. These increase the stress on the network by further dividing monolithic tasks into several subtasks that will have to communicate with each other [12].

Usually, when a job arrives, the orchestrator tries to optimize the data center resources and decides on which servers the job's tasks should be executed. Traditionally, this is done using scheduling algorithms that 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 the 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* that takes into account the limited communication bandwidth. In this framework, traditional (CPU) tasks stand alongside new *network tasks*. As usual, servers execute (CPU) tasks, but *network machines* execute network tasks, 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 that belong to the same job, we would only incur a communication delay if T_1 and T_2 are scheduled on two different CPU machines. In such a case, we would have to schedule a network task $T_{1\to 2}$ 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 that 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 finding the *minimum makespan* of a set of tasks over all possible schedules, where the makespan of a schedule is the maximum over all tasks of their completion times. The problem is NP-complete and we show that the simple 3-approximation List Scheduling algorithm with communication delays

A short version of this work appeared in [1].

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 executed. We provide provably efficient 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 [13]. We show that our algorithms are very efficient for scenarios in which network capacity has to be taken into account.

A short version of this work has been publisThe 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 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 to improve job performance.

In [8], 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 their approach can reduce the shuffle phase duration by a factor of 1.5.

A family of works is based on the coflow abstraction [14], that is, a collection of parallel flows belonging to the same job. Several coflow schedulers were proposed. Varys [9] is a coordinated coflow scheduler designed with the goal of maintaining high network utilization and guaranteeing starvation freedom. Chowdhury et al. [8, 9] proposed centralized coflow schedulers.

A decentralized solution is presented in [10]. 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 them to reduce both the average and the tail task completion times.

Luo et al. [15] study the problem of average coflow completion time from a theoretical point of view. They first show that minimizing the average completion time for a given set of coflows is NP-hard. More precisely, they show that minimizing the total coflow completion time is equivalent to the problem of minimizing the sum of completion times in a concurrent open shop, a well-known NP-Hard problem. Then, they develop a 2-approximation coflow schedule algorithm for the offline case.

In [16], the authors propose OFScheduler, a dynamic network traffic optimization method based on OpenFlow with the goal of improving the computational efficiency of MapReduce jobs in heterogeneous clusters. They show that thanks to their approach both the bandwidth utilization and the performance of MapReduce jobs can be improved.

In [17], 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 [18] 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 on a small number of racks to increase 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., [19] for a comprehensive survey.

Scheduling with *precedence constraints* or *list scheduling* was introduced in [20]. 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, communication was introduced into the family of problems called *scheduling with communication delays*. If a task T produces some data that is used as an input by a second task T', and if both tasks are not executed on the same machine, it takes a delay d_{ij} to send this 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 [21]. When the communication costs are further simplified to be unitary, a 2-approximation exists without the above additional hypothesis [22].

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*) that have to be executed on m machines (also called processors or servers in the literature). A machine M_j has a processing

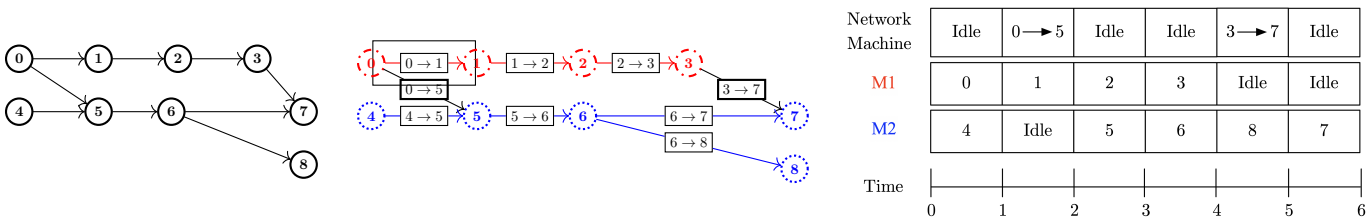

Fig. 1: (Left) The dependency (di)graph of a job J with 9 (CPU) tasks. (Middle) Modeling with network tasks indicated with rectangles. We provide a feasible schedule of job J. Scheduled graph: (CPU) tasks executed by Machine 1 are in dashed red, by Machine 2 in dotted blue, carried out network tasks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

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 execution time of a task T_i on the machine M_j is $p_{i,j} = s_i/S_j$.

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. The layer of a task T in a DAG, $\text{layer}(T)$, is defined as

$$\text{layer}(T) = \begin{cases} 1 & \text{if } d^-(T) = 0 \\ \min\limits_{T' \in N^-(T)} \text{layer}(T') & \text{otherwise} \end{cases}$$

When the task T_i has completed its execution, it may have computed data that are needed to execute task T_j. We model this by introducing a *network task* $T_{i\to j}$ that 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 execution time of a network task $T_{i\to j}$ on the network machine N_ℓ is $p_{i\to j,\ell} = s_{i\to j}/C_\ell$.

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 that may be executed. They are placed on machines M_1 and M_2 respectively. Their execution 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 $T_{0\to 5}$ is executed by the network machine. It takes one time unit (size/capacity). The makespan of this schedule is thus 6.

B. Model Discussion

On Parallel Execution and Bandwidth Sharing. First, note that using an indivisible resource to execute two tasks, say T and T', in parallel is suboptimal in the sense that there will always exist a schedule with a makespan equal to or less than T that executes first T and then T' on the same resource (the advantage of sequential execution is that one of the tasks is finished first, and the next dependent task could be executed on another resource). Note also that when considering the classical alternative goal of minimizing the average execution time, not doing parallel execution is strictly better. The same observation can be made for network machines. Doing bandwidth sharing on an indivisible resource is suboptimal, because the data for one of the tasks would arrive first if the data were sent sequentially over the link.

However, real systems make extensive use of parallel execution and bandwidth sharing. This is indeed efficient for divisible resources, such as machines with multiple cores, optical networks with multiple wavelengths, or wireless networks with multiple channels. These cases can be easily modeled in our framework by defining one machine per subresource. For example, for a multi-core processor, each core is represented by a machine, and all cores of the same machines are connected by network machines with infinite capacity (or the capacity of their internal buses). Similarly, a wavelength or channel can be modeled by a network machine. We can also go one step further if we want to do parallel execution or bandwidth sharing (at the cost of potentially lower efficiency) on a subresource by modeling the latter as multiple machines.

Offline vs. Online Scheduling. We have considered a classic offline scheduling scenario in which we want to schedule a set of jobs with full information. However, in a large number of practical scenarios, jobs arrive over time. This is classically handled by defining short time windows in which the scheduler waits for a batch of jobs, and then schedules that batch of jobs at the end of the window. Our modeling and algorithms apply to this scenario. We leave as future work the study of online scheduling algorithms with network tasks, for the scenario where a new job must be scheduled as soon as it arrives.

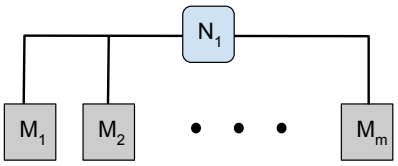

Fig. 2: Modeling the communications between m machines connected through a bus. A single network machine, N_1, is used.

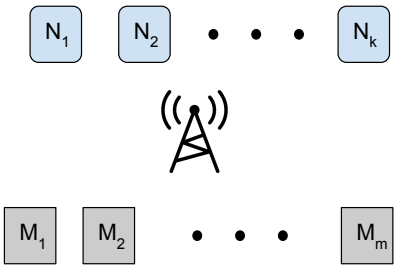

Fig. 3: Modeling the communications between m machines connected through a wireless antenna with k channels. One network machine is used per channel.

C. 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 (see Figure 2) or connected via an antenna (WiFi, 4G, etc.) using a model with one network machine per channel (see Figure 3). However, more complex networks can be represented.

Data center networks. The *simplicity of the model lies in the fact that only border links (i.e., links between the servers and the Top of Rack (ToR) switches) have to be modeled*. Indeed, data center architectures are built with large bisection bandwidth [23], or with full bisection bandwidth in the case of topologies such as Fat Trees or VL2. In fact, they are permutation networks: when the border has enough capacity to send and receive a communication, a path with enough capacity for the communication exists inside the network. Thus, only border links have to be taken into account.

We consider a data center with m servers, see an example in Fig. 4. 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 [24]. Each server is modeled by a machine. However, we now introduce two network machines per link connecting a 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

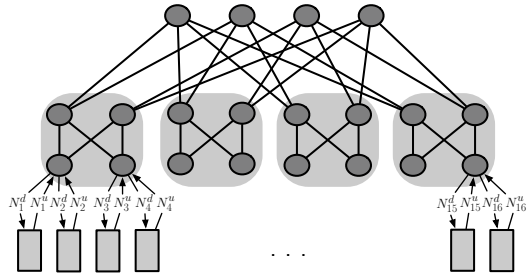

Fig. 4: Modeling data center communications with network machines for a 4-Fat Tree with 16 (racks of) servers.

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, $p_{ij} = p_i$ for all tasks $T_i \in \mathcal{T}$ and $p_{i\to j,\ell} = p_{i\to j}$ 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., [25]. 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 of the problem of scheduling with precedence. However, there exists a simple greedy algorithm that has a 2-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).

List-Scheduling

Next, we study the performance of the well-known "List Scheduling" algorithm that is 2-approximate in the case of infinite network capacity (see [22]). 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 [22] provides a 2-approximation of the problem *scheduling with communication delays* when (CPU) task execution 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) that

Algorithm 1 Generalized List scheduler

```
U = T                                   ▷ U is the set of unprocessed tasks
t = 0                                   ▷ t is the clock
while U ≠ ∅ do
    t = t + 1
    for p = 1, 2, ..., m do             ▷ Iterate on machines
        Compute the set of available tasks A_{p,t}
        if A_{p,t} ≠ ∅ then
            min = {T' ∈ A_{p,t} | T' ⊏ T for all T ∈ A_{p,t}}
            Allocate to machine M_p the task min at time slot (t − 1, t]
```

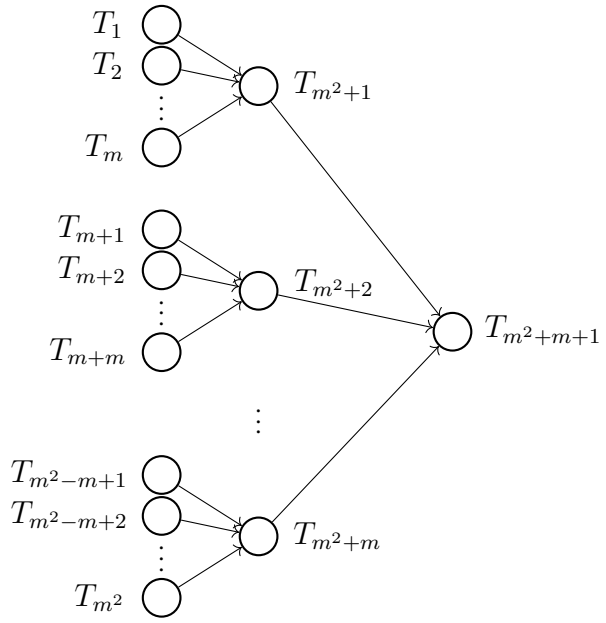

Fig. 5: Instance of the proof of Theorem 1.

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 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 that 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, see Figure 5. 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 $\mathcal{M}(S^*) = 2m + 1$.

On the other hand, Algorithm 1 may choose an order such that the tasks $T_{(i-1)m+1}, T_{(i-1)m+2}, \ldots, T_{im}$ are simultaneously scheduled on the 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 $T_{m^2+2}, \ldots, T_{m^2+m}$ 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 that 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, Generalized List Scheduling, is a generalization of the well-known List Scheduling algorithm. The second one, Partition, 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. To this end, we generalize the notion of an available task defined for the UET list scheduler algorithm [22]. The goal

Algorithm 2 GENERALIZED LIST SCHEDULING (G-LIST)

```
U = T                                   ▷ U is the set of unprocessed tasks
t = 0                                   ▷ t is the clock
while U ≠ ∅ do
    t = t + 1
    compute A the set of available task/machine-assignments.
    sort A according to number of needed communications
    while A ≠ ∅ 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
Test all possible placements of the last 2 tasks, choose the one with the minimum makespan.   ▷ Feature (3)
Solve the reverse workflow of T                                                                ▷ Feature (2)
```

is then to avoid carrying out unnecessary 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 pairs (available task, machine) at time t and sort them according to the number of required communications to be performed by the schedule. The algorithm thus places a task on a machine in which the most data are available if possible. (2) A task is placed on a machine at time t only if all its communications tasks can be performed before time t. If this is not possible, 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 M 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 number of needed communications to be scheduled (and, then, by machine index[1] for tasks with the same number of communications, with M_1, the machine executing the first task). While A is not empty, G-LIST 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 anymore.

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 execution times of tasks are large, it is not necessary to iterate over all the time slots. Several features are added to improve the algorithm:

[1]It avoids using a new machine if a machine already used in the past is available.

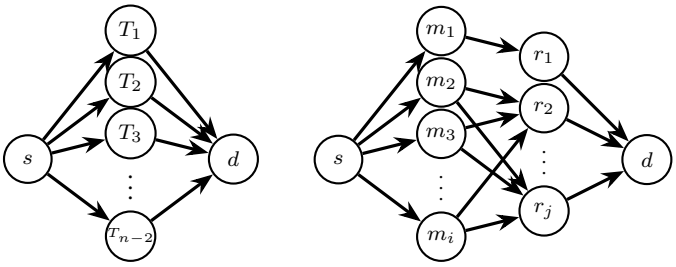

Fig. 6: Example of simple (left) and single-stage (right) MapReduce workflows with i map tasks and j reduce tasks.

(1) In the 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 [20].
(2) The algorithm performs two passes. The first one considers the workflow, and the second one considers the "reverse workflow", in which an arc between tasks T_i and T_j is transformed into an arc between tasks T_j and T_i. Then, the better between the two passes is selected (see the result of Lemma 1). The idea is that the first pass optimizes out-trees and the second pass the in-trees, in the sense that tasks in the same subtrees are placed on the same machine if possible.
(3) For the last two tasks, we test placing it on every machine and we chose the solution with the minimum makespan.

We call G-LIST-BASIC the variant of the algorithm that does not have feature (3). We also define another variant of the algorithm, G-LIST-MASTER, as follows. 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 to see if it would be faster to place it on the master machine once it is free. In other words, we only place a task on a slave machine if its completion time plus the time to send the result back to the master machine is less than the completion time of all remaining tasks on 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* [20], and when considering dependencies and communication (but no bandwidth limit), to List Scheduler [22].

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. Such a workflow is sometimes also referred to as *Diamond topology* in the literature, see e.g. [26]. 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. 6. The execution times of tasks $T_1, \ldots, T_{n-2}$ are all equal to p_b. Similarly, the tasks $T_{s\to T_i}$ (resp. $T_{T_i\to d}$), for $1 \leq i \leq n-2$, share the same communication time, p_a (resp. p_c). We also note p_s and p_d the execution times of s and d, respectively. 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 [27].

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 $T_1, \ldots, T_{n-2}$ 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 $k \geq 3$ [28].

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

Proof. **Definitions.** Let M_s (resp. M_f) be the machine on which the initial (resp. final) task s (resp. d) is executed. We note n_s (resp. n_d) the number of tasks executed on M_s (resp. M_f).

First, note that, in addition to the initial machine M_s, G-LIST uses a maximum of $\lceil p_b/p_a \rceil$ machines. Indeed, the time to send one task to each of the $\lceil p_b/p_a \rceil$ machines is at least $p_a\lceil p_b/p_a \rceil \geq p_b$. Thus, within this time, one of these machines is available to execute a new task.

In the following, we assume that $p_a \geq p_c$. Indeed, when $p_a < p_c$, the reverse workflow has the desired property and has the same minimum makespan, see Lemma 1. As G-LIST considers both the workflow and its reverse (see (2)), if it solves optimally one of them, it is optimal for both.

Lemma 1. *Let W be a workflow and R its reverse workflow. To every schedule of R corresponds a schedule of W with the same makespan. W and R thus have the same mimimum makespan.*

Proof. Let S_R be a schedule of R of makespan $\mathcal{M}(R)$. Then, the schedule S_W for which the starting time of task T is $\mathcal{M}(R) - c_T$, where c_T is the completion time of T in S, is a feasible schedule for W with the same makespan as W. Indeed, in S_R a task T finishes at time c_T before its out-neighbors (which are in-neighbors in W) executions start. Thus, in S_W, a task T starts at time $\mathcal{M}(R) - c_T$ after all its in-neighbor executions have finished. The initial machine in S_R is the final machine in S_W, and vice versa. Moreover, sending an output from a machine M to the final machine in W takes the same time as sending the input from the initial machine to M in R, since only the sense of an arc is reversed in R (and vice versa). □

We now derive some results for G-LIST-BASIC, Lemmas 2, 3, and 4.

Lemma 2. G-LIST-BASIC *is near optimal on simple MapReduce workflows with $a \geq c$, meaning that:*

$$\mathcal{M}_{Basic} \leq OPT + p_c,$$

where $\mathcal{M}_{Basic}$ is the makespan of the schedule given by G-LIST-BASIC *and OPT the minimum makespan over all possible schedules.*

Proof. Consider the subworkflow, W, without the final task T_d. Its minimum makespan, OPT_W, is smaller than the one of the original simple MapReduce workflow, OPT. If fact, we have: $OPT \geq OPT_W + p_d$, as all tasks of W have to be executed before T_d can start.

The minimum makespan of W is achieved by G-LIST-BASIC, as no resource can be used in a faster way. Indeed, (1) The initial machine works continuously until all tasks are scheduled. (2) If $m \geq \lceil p_b/p_a \rceil$, its upload bandwidth is continuously used, and the $\lceil p_b/p_a \rceil$ slave machines execute their tasks as soon as they receive their inputs. (3) If $m < \lceil p_b/p_a \rceil$, the $m-1$ slave machines work continuously after having started to execute their first task, until all tasks are scheduled.

When all tasks of W have been executed, as $p_a \geq p_c$, it is possible to send all outputs computed on the slave machines to the initial machine within the time $OPT_W + p_c$. Indeed, all tasks executed on a slave machine start as soon as their input is received, so their start (and, thus, their completion time) are distant by a time $p_a \geq p_c$. Thus, the output of each of these tasks can be sent to the master machine before the end of the next completion time of the next slave task. The output of the last executed slave task is then sent within a time p_c.

Thus, the initial machine can complete the last task T_f in time at most $OPT_W + p_c + p_d$. The algorithm will select the first machine being able to finish task T_f, thus $\mathcal{M}_{\text{Basic}} \leq OPT_W + p_c + p_d$, leading to:

$$\mathcal{M}_{\text{Basic}} \leq OPT_W + p_c + p_d \leq OPT + p_c.$$

□

Lemma 3. *The solution provided by* G-LIST-BASIC *cannot be improved by executing one fewer task on M_i.*

Proof. At the end of the execution of G-LIST two cases are possible.
(1) T_{n-2} is executed on M_s. In this case, T_d is also executed on M_s. Indeed, the same number of or more tasks are executed on M_s. Thus, $T_{\text{Basic}} = \min(c_{n-2}, c_{n-3} + p_c) + p_d)$. Executing the task T_{n-2} on a machine $M \neq M_s$ would give a makespan of $\mathcal{M}(S') \geq c_{n-3} + \max(p_a - p_b, 0) + p_b + p_d$. If $p_a \geq p_b$, $t_{S'} \geq c_{n-3} + p_a + p_d \geq c_{n-3} + p_c + p_d \geq T_{\text{Basic}}$. If $p_a < p_b$, $t_{S'} \geq c_{n-3} + p_b + p_d \geq c_{n-3} + p_c + p_d \geq T_{\text{Basic}}$.
(2) T_{n-2} is executed on a machine $M \neq M_s$. We have two cases: if T_d is executed on M, $t_S = \max(c_{n-3} + c, c_{n-2}) + p_d$ and if T_d is executed on M_s, $t_S = c_{n-2} + p_c + p_d = T_{\text{Basic}} + p_c + p_d$. Executing an additional task on M makes that the makespan is $\mathcal{M}(S') \geq \max(c_{n-5} + p_c, c_{n-2} + \max(p_a - p_b, 0) + p_b) + p_d$. If $p_b \geq p_a$, $c_{n-2} + \max(p_a - p_b, 0) + p_b) = c_{n-2} + p_b \geq c_{n-2} + p_c \geq c_{n-3} + p_c \geq c_{n-5} + p_c$. It gives,

$\mathcal{M}(S') > \mathcal{M}(S)$. Now, if $p_a > p_b$[2], $c_{n-2} + \max(p_a - p_b, 0) + p_b) = c_{n-2} + p_a \geq c_{n-2} + p_c \geq c_{n-3} + p_c \geq c_{n-5} + p_c$. It gives again, $\mathcal{M}(S') > \mathcal{M}(S)$. □

Lemma 4. *When $p_a \geq p_c \geq p_b$, an optimal solution S_{opt} can be found from the solution S given by Basic* G-LIST*:*

(i) by executing T_{n-2} and T_d on M_s in the case T_d is executed on M_2 in S.

(ii) by setting $S_{opt} = S$ or by executing T_{n-2} on M_s in the case T_d is executed on M_s in S.

Proof. First, Lemma 3 tells us that the only way to improve the solution of G-LIST is to execute more tasks on M_s. Therefore, the best solution found by doing so is optimum.

This proposition is trivially true when $m = 1$, so in the following, we suppose that there is more than one machine. When $a \geq b$, only one additional machine, M_2, is used, because $\lceil p_b/p_a \rceil = 1$. We note $\mathcal{M}(S)$ the makespan of S.

(i) Note that T_{n-2} cannot be executed on M_s, otherwise, it would be preferable to execute T_d on M_s. Thus, the last task executed on M_s (resp. M_2) is T_{n-3} (resp. T_{n-2}). Since the output of T_{n-3} has to be sent to M_2, we have that $\mathcal{M}(S) \geq (\max c_{n-3} + p_c, c_{n-2}) + p_d$, . Now, Consider the solution S' built from S, in which T_{n-2} and T_d are executed on M_s instead. Its makespan, $\mathcal{M}(S')$, is such that $\mathcal{M}(S') = \max(c'_{n-2}, c_{n-4} + p_c) + p_d$. Now, we have $c'_{n-2} = c_{n-3} + p_b \leq c_{n-3} + p_c$ and $c_{n-4} + p_c = c_{n-2} - p_b - (p_a - p_b) + p_c = c_{n-2} + p_c - p_a \leq c_{n-2}$. It gives, $\mathcal{M}(S') \leq \mathcal{M}(S)$. The solution is optimum because executing an additional task on M_s would increase the makespan. Indeed, if we execute T_{n-4} on M_s, the makespan $\mathcal{M}(S'')$ would be $\mathcal{M}(S'') = \max(c_{n-3} + 2p_b, c_T) + p_d$, with T the last job executed on M_2. Now, we have that $c_T < c_{n-4} = c_{n-2} - p_a + p_b$. Additionally, $c_{n-3} > c_{n-2} - p_b$. Using these two expressions, we get that $c_{n-4} + p_a < c_{n-3} + 2p_b$. Since $p_a \geq p_c$, we have that $c_T < c_{n-4} + p_c < c_{n-3} + 2p_b$. Therefore, the makespan $\mathcal{M}(S'') \geq \mathcal{M}(S') = \max(c_{n-3} + p_b, c_{n-4} + p_c) + p_d$. Thus, S' is optimal.

(ii) In this case, the solution is improved when $c_T + p_c \geq c_{n-2}$, where c_T and c_{n-2} are the completion times of the last tasks executed on M_2 and M_s, respectively. Again, it can be proven that adding an additional task to M_s would only increase the makespan.

Note that, for all optimal solutions, the first and last tasks are executed on the same machine M_s. □

Lemma 5. *When $p_a \geq p_c \geq p_b$,* G-LIST-MASTER *is optimal on simple MapReduce workflows.*

Proof. This result is a direct corollary of Lemma 4. Indeed, G-LIST-MASTER finds S_{opt}, as it tests, before placing a task on a slave machine, whether it would not be faster to place it on the master machine when it will be free. This is exactly the condition tested in (ii) to find S_{opt}. This also prevents it to place T_{n-2} on M_2 for the case (i), leading again to S_{opt}. □

[2]Note that when $p_a > p_b$, the upload bandwidth of M_s is executed at maximum speed until the beginning of the execution of T_{n-2} in the solution S, unless if $m = 1$, in which case, the proposition is trivially true.

Corollary 1. *When $p_a \geq p_c \geq p_b$,* G-LIST-MASTER *is optimal on simple MapReduce workflows.*

Proof. This result is a direct corollary of Lemma 4. Indeed, G-LIST tests all positions for the last two tasks (i.e., T_{n-2} and T_d here) by Feature (3), so in particular the optimum solutions of the previous lemma. □

Lemma 6. G-LIST *is optimal on simple MapReduce workflows with $p_b \geq p_c$ and $p_a \geq p_c$.*

Proof. As $p_a \geq p_c$, according to Lemma 2, $OPT \leq OPT_W + p_c$. Consider the optimum schedule for W. In this schedule, it is impossible to execute a task before its starting time. Therefore, the only way to improve the (Master Basic) G-List schedule is to execute a task at a later time on a different machine to avoid transmitting data. This would only be useful if the task were executed on the machine M_f carrying out T_d. Note that moving two tasks to M_f would lead to a solution worse than that of $G - List$. Indeed, the makespan would be greater than or equal to $OPT_W + p_b$, because the completion time of the last task in the schedule is greater than $OPT - p_b$. Otherwise, the last task of W would have been executed on this machine. Since $p_b \geq p_c$, the makespan would exceed $OPT_W + p_b \geq OPT_W + p_c \geq \mathcal{M}_{\text{Basic}}$. Thus, moving only one task to another machine may be useful. G-LIST is in fact testing this by checking all potential machines to execute the one before last task, thanks to Feature (3). □

We can now conclude the proof of Proposition 1. G-LIST is optimal when $p_b \geq p_c$ and $p_a \geq p_c$ by Lemma 6. G-LIST-MASTER is optimal when $p_c \geq p_b$ and $p_a \geq p_c$ by Corollary 1. Last, since G-LIST tests the reverse workflow (Feature (2)), Lemma 1 gives the optimality when $p_c \leq p_a$. □

C. PARTITION

When the workflows are complex, the greedy algorithm may have difficulty allocating 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 executed by the machines, while minimizing the network tasks that would be need to be performed. 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) that 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 [29]. 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 [29, 30]. The approximation algorithm with the best-known approximation factor, $O(\sqrt{\log n \log k})$, is due to Krauthgamer et al. [31].

PARTITION-ASSIGN 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 k-balanced partitioning problem, 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).

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 executed 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 $W^*_{CPU} \leq n$, the total amount of work to be executed.

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. Then, 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 $S_A = \max\_part\_size + \text{cut\_weight}$, with cut_weight the capacity of its cut and $\max\_part\_size$ the maximum weight of a part.

On one hand, we have that

$$\text{cut\_weight} \leq O(\sqrt{\log n \log k}) W^*_N.$$

Indeed, S^* provides a solution for the k-balanced partitioning algorithm, as it uses at most k machines. On the other hand, we have that

$$\text{max\_part\_size} \leq \frac{2n}{k}.$$

As $\frac{n}{k} \leq W^*_{CPU}$, we get that max_part_size $\leq 2W^*_{CPU}$. This yields that $S_A \leq O(\sqrt{\log n \log m}) S^*$. □

Algorithm 3 PARTITION-ASSIGN

1: **Input:** Set of workflows G, m number of machines.
2: partitions[m] ▷ Solutions of m partitioning problems
3: **for** $k = 1, 2, \ldots, m$ **do** ▷ Iterate on number of processors
4: partitions[k] ← Compute an approximate solution of the k-balanced partitioning problem for G.
5: 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 [30]. 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 [32] to solve bisection, leading to a time complexity of $O(mn^3 \log n)$.

2) SCHEDULING WHEN PLACED*:* Once the decision in terms of which tasks to be performed on which machines has been performed by the partition algorithm, PARTITION-ASSIGN, it remains to decide in which order to carry them out. The problem to be solved while minimizing the makespan is the SCHEDULING WHEN PLACED problem. We first discuss the hardness of the problem and then provide an approximation algorithm to deal with it.

Theorem 3. *The* SCHEDULING WHEN PLACED *problem is NP-complete and cannot be approximated within a factor* $\frac{5}{4}$*. Moreover, no asymptotic Polynomial-Time Approximation Scheme (PTAS) exists for the problem.*

Proof. The proof is done by a reduction from the 3-SAT problem to SCHEDULING WHEN PLACED. We consider the variant of 3-SAT in which each clause has exactly 3 literals. Let $f = C_1 \wedge \cdots \wedge C_m$ be a boolean formula with m clauses and n variables, $x_1, \ldots, x_n$. We note l^1_i, l^2_i, and l^3_i the 3 literals of the clause C_i and we have $C_i = l^1_i \vee l^2_i \vee l^3_i$. We associate to f the following instance $\mathcal{I}$ of the SCHEDULING WHEN PLACED problem. We associate a machine to each clause (called "clause" machine in the following) and to each variable ("variable" machine), giving $n + m$ machines. We note M_{C_i} (resp. M_{x_i}) the machine associated to C_i (resp. x_i), see Fig. 7. We then define (i) a task per literal in f, placed in the machine corresponding to the clause in which it appears, and (ii) two tasks per variable, for its positive and negative literals, placed in the machine of the variable. $\mathcal{I}$ thus has $3m + 2n$ tasks. All tasks take unit time. The machines M_{C_i}, for $1 \leq i \leq m$, have to execute 3 tasks and the machines M_{x_i}, for $1 \leq i \leq n$,

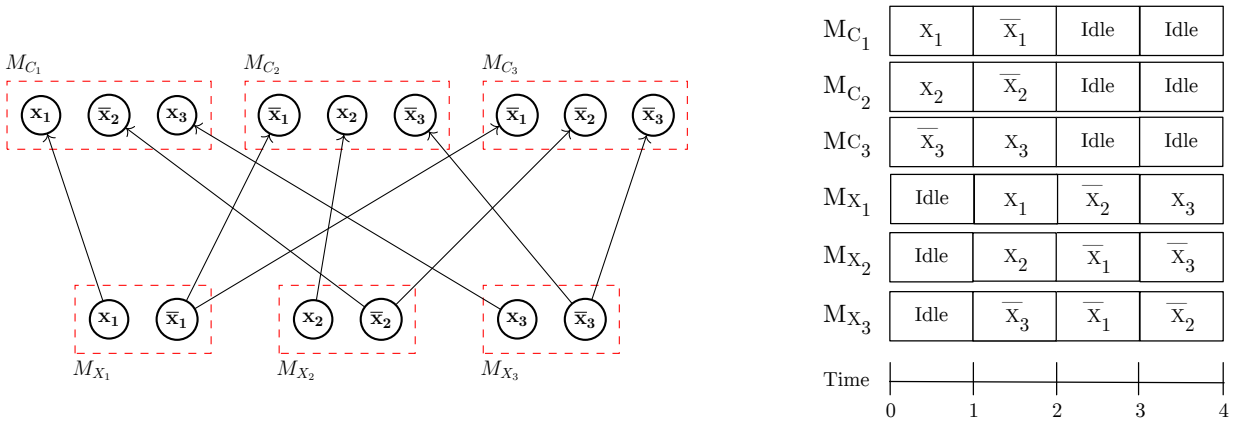

Fig. 7: (Left) Construction used to prove the hardness of the SCHEDULING WHEN PLACED problem. Example for $f = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_4 \vee \bar{x}_3 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_4)$. (Right) Corresponding timeline.

2 tasks. We then introduce jobs defining the dependencies between tasks. We define a job per literal, l, present in f. Consider $l = x_i$ (resp. $l = \bar{x}_i$), the workflow links the task x_i in the machine M_{x_i} to all the tasks x_i in the machines M_{C_j}. This implies that the task in machine M_{x_i} has to be executed before the tasks of the "clause" machine.

We now show that the formula f can be satisfied if and only if the makespan of $\mathcal{I}$ is 4.

If the formula can be satisfied, there exists a truth assignment for which one of the literals of each clause is true. We define the schedule in which the task of the true literals is executed first in the "variable" machines. Note that all will be executed in the first time step. The false literals will then be executed in the second time step. For the "clause" machines, we schedule first the tasks of the true literals (in any order) and then the tasks of the false literals (in any order). During the second time step, we know that, for each "clause" machine, there exists at least one task that can be executed (a task of a true literal) as the formula is satisfied. At the third time step, any task of a "clause" machine can be executed as all tasks of the "variable" machine have been finished. The scheduling will thus finish after 4 time steps.

Conversely, if there exists a schedule with makespan 4, this implies that each "clause" machine is working during time step 2. It means that there exists at least a task of a literal of the clause in a "variable" machine that has been executed during time step 1. Consider the truth assignment in which the corresponding literal is set to true. This truth assignment satisfies f.

Note that the proof directly implies that the SCHEDULING WHEN PLACED problem cannot be approximated within a constant factor $\frac{5}{4}$. Indeed, if the formula f cannot be satisfied, then the scheduling takes 5 time steps. Hence, deciding if the makespan is 4 or 5 is NP-hard.

More generally, deciding if the makespan is $4T$ or $5T$ for any $T \in \mathbb{R}_+^*$ is NP-hard. We prove it by taking T copies of our construction and connecting all the sinks of the k-th copy to the sources of the $(k+1)$-th copy, for $1 \leq k \leq T-1$. Doing so, the tasks of the $(k+1)$-th copy cannot start before all the tasks of the k-th copy have been finished. If f can be satisfied, the makespan of the new construction is $4T$, and $5T$ otherwise. SCHEDULING WHEN PLACED thus does not have asymptotic PTAS. □

Algorithm 4 PARTITION-SCHEDULE

Input: Set of workflows G with the corresponding set of tasks $\mathcal{T}$. The lists $Assigned(p)$ of the tasks of $\mathcal{T}$ assigned to machine p, for $p = 1, \ldots, m$.
$U = \mathcal{T}$ ▷ 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 $A_{p,t}$ from $Assigned(p)$
    if $A_{p,t} \neq \emptyset$ **then**
      $T \leftarrow$ a task such that $layer(T) \leq layer(T')$ for all $T' \in A_{p,t}$
      Allocate the task T to machine M_p at *time slot* $(t-1, t]$ and the needed network tasks.

We now propose an algorithm, PARTITION-SCHEDULE, to solve the problem. The algorithm 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, see Algorithm 4. PARTITION-SCHEDULE is efficient in practice as the depth of workflows usually is small.

Theorem 4. PARTITION-SCHEDULE *provides a* $\mathrm{depth}(W)$*-approximation algorithm of the* SCHEDULING WHEN PLACED *problem, where* $\mathrm{depth}(W)$ *is the depth of the workflow* W *to be scheduled.*

Proof. Consider an optimal schedule S^* and let $\mathcal{M}(S^*)$ be its makespan. We denote by $p(L_i)$ the time to process the tasks of layer i and by $p(L_i \rightarrow L_j)$ the time to process the network tasks between layers i and j. Clearly, $p(S^*) \geq p(L_i)$ for $1 \leq i \leq \mathrm{depth}(W)$. Similarly, $p(S^*) \geq p(L_i \rightarrow L_{i+1})$

for $1 \leq i \leq \text{depth}(W) - 1$. In addition, the makespan of PARTITION-SCHEDULE, $\mathcal{M}(A)$, is such that

$$\mathcal{M}(A) \leq \sum_{i=1}^{\text{depth}(W)} p(L_i) + \sum_{i=1}^{\text{depth}(W)-1} p(L_i \rightarrow L_{i+1}).$$

We thus have $\mathcal{M}(A) \leq (2\ \text{depth}(W) - 1)p(S^*)$. □

We thus obtain a two-phase algorithm, PARTITION. It first solves PARTITION-ASSIGN to decide which task to be performed on which machine. Then, it uses PARTITION-SCHEDULE to schedule the tasks over time.

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 [13]. 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 [22], which was proposed to handle communication delays, but 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 the 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 [33].

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 the *network factor*, as 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 [8].

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. 6). 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 depends on a given task in 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 0.2, 0.4, and 0.4, respectively. It corresponds to a realistic distribution as at least 50% of applications in clusters can fit in the MapReduce paradigm [27]. 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, the number of tasks in a job, the workflow edge density, and thethe 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 normalizes 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. 8). 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 the executed. When the number of machines increases, the ratio increases as well since the tasks are performed 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 (i.e., 1-Stage MapReduce and random), the ratio has similar or lower values, showing the efficiency of our algorithms.

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 workflows (Fig. 8a), 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. 8b), PARTITION is a lot more efficient than G-LIST as this type of workflow is more complex to schedule. On random workflows, both

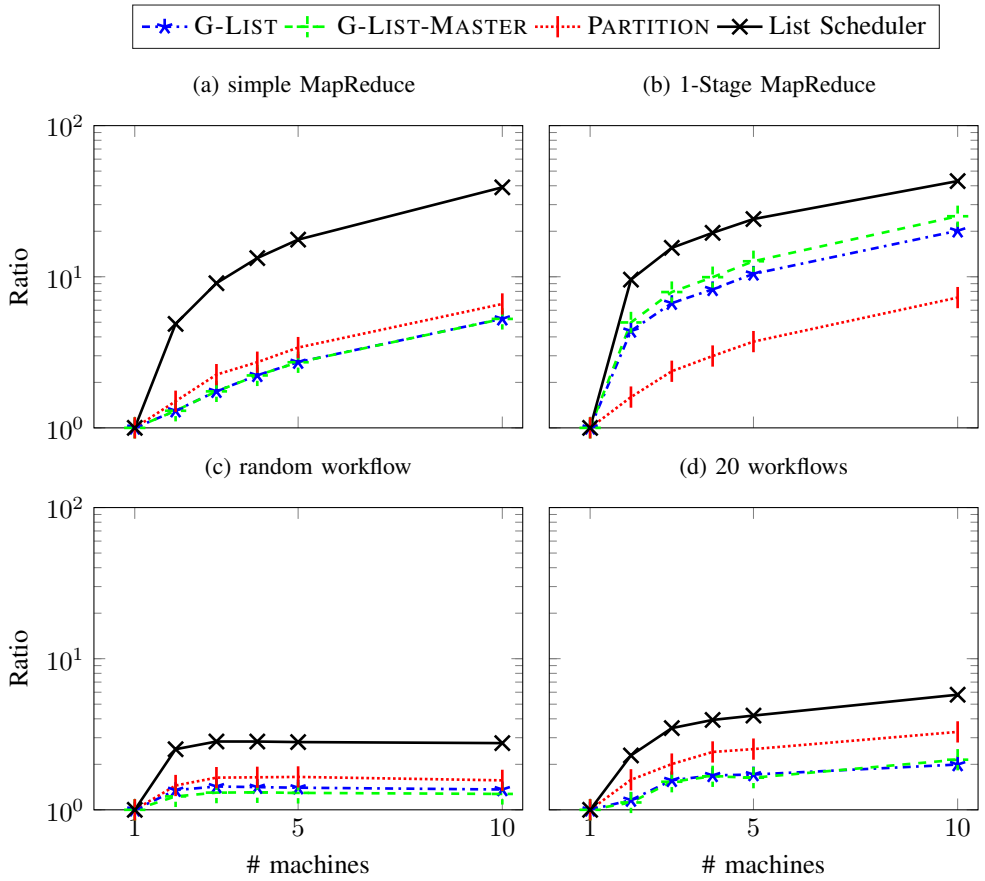

Fig. 8: Efficiency of the proposed algorithms as a function of the number of machines for different types of workflows.

algorithms behave well. Indeed, the ratio is close to 1 in this case. The reason is that random workflows have DAGs with long depths on average, and so, fewer scheduling combinations exist (see the following discussion for edge density). On the sets of 20 workflows, G-LIST and G-LIST-Master have the best performance.

Workflow size.

We observe similar results when varying the *number of tasks per workflow* (Fig. 9). For simple MapReduce workflows, PARTITION behaves almost as well as G-LIST, which is optimal for such workflows. However, PARTITION is significantly better than G-LIST for the more complex MapReduce workflows. For random workflows, the algorithms perform similarly with a small advantage for G-LIST-Master on small workflows and for PARTITION for larger ones. Finally, for the set of 20 workflows, G-LIST and G-LIST-Master are very efficient, as they reach ratios close to 1.

To better understand the characteristics of the workflows for which each algorithm is more efficient, we studied two parameters: edge density and network factor.

Edge density. We made the edge density vary from 0 to 1 (Fig. 10). 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. 10a). For random workflows (Fig. 10b), 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. 11). When the network factor is zero, all algorithms are equivalent. Indeed, 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 workflows 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

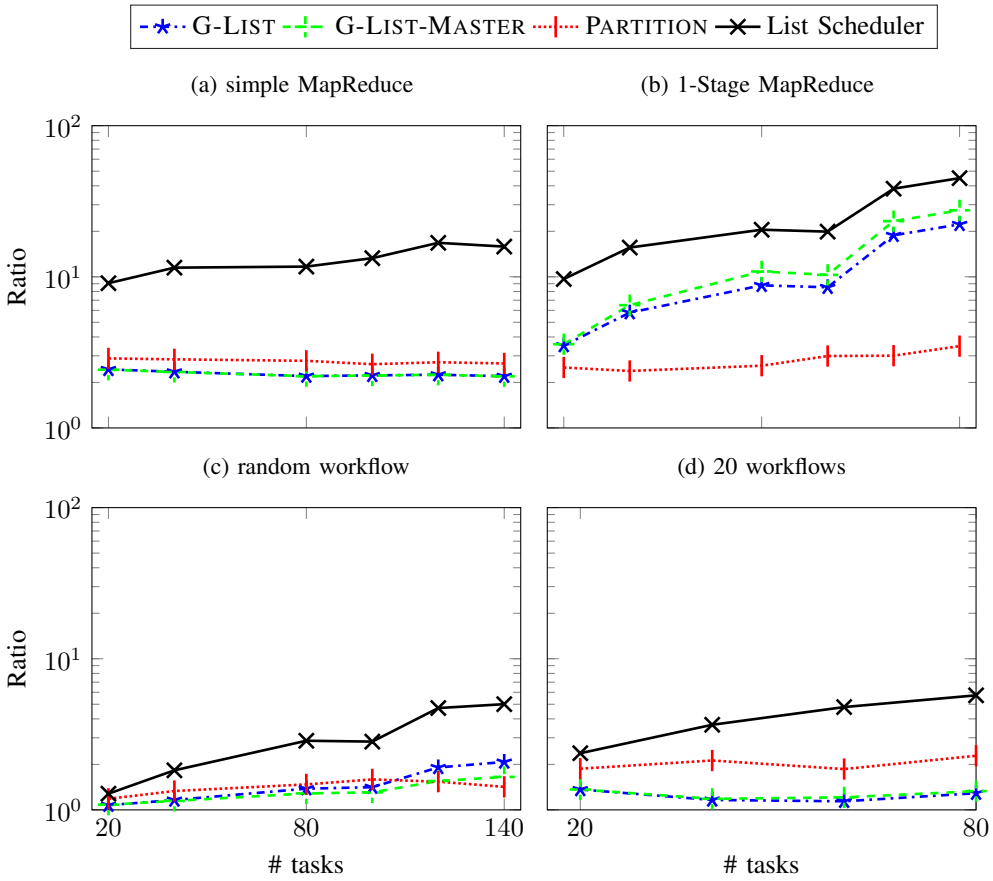

Fig. 9: Efficiency of the proposed algorithms as a function of the number of machines for different types of workflows.

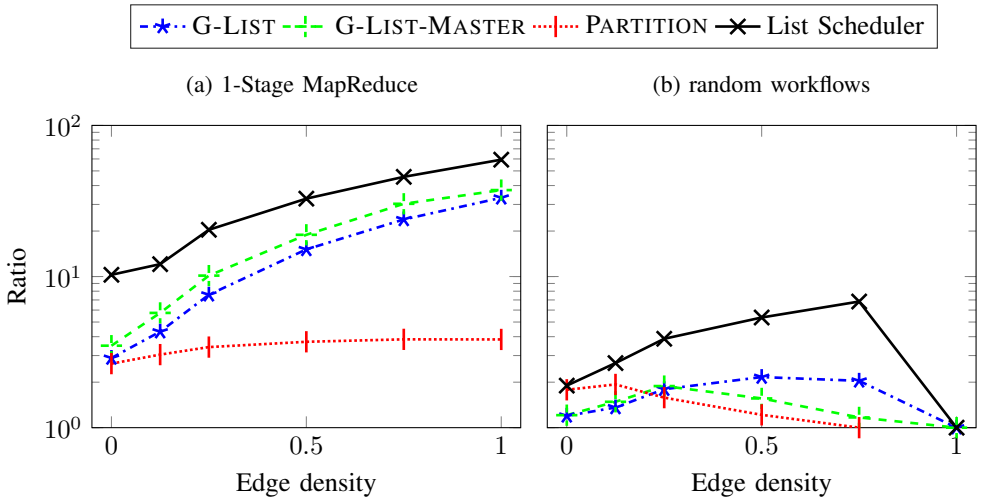

Fig. 10: Efficiency of the proposed algorithms as a function of the workflow edge density.

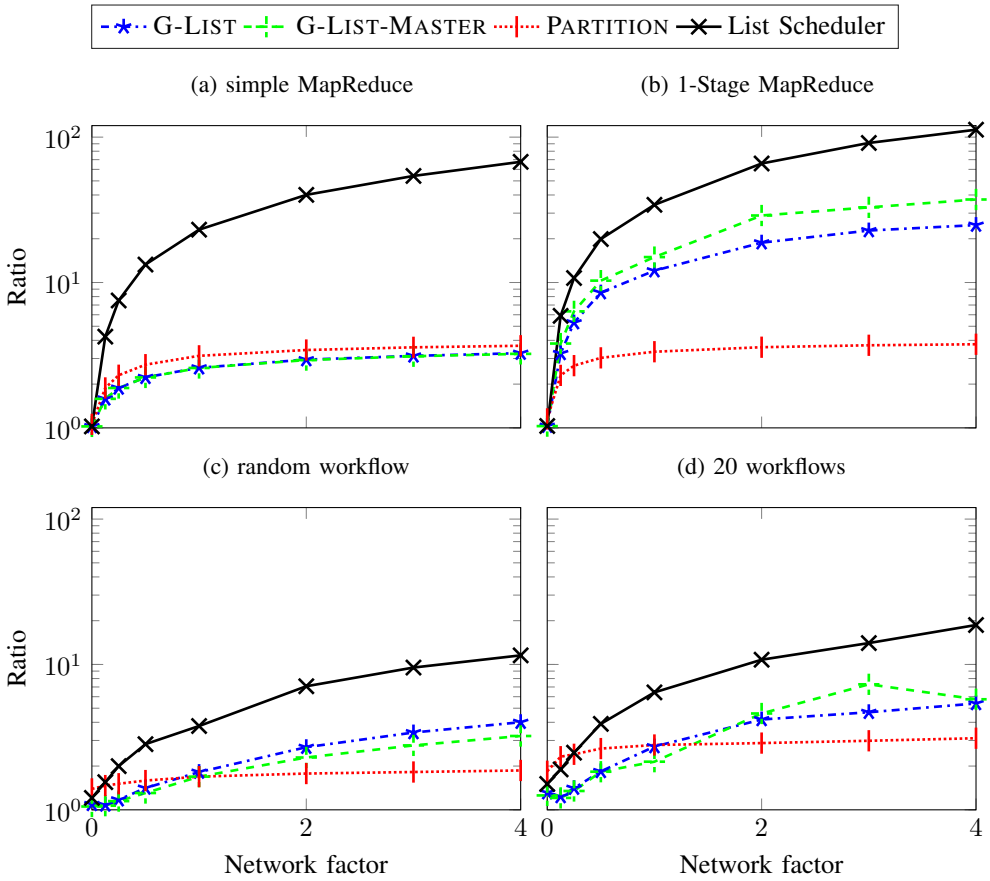

Fig. 11: Efficiency of the proposed algorithms for different network factors and types of workflows.

of applications and network capacity.

VII. Conclusion

In this paper, we proposed a new framework to model the orchestration of tasks in a data center for scenarios in which the network bandwidth is a limiting resource. We introduced a new problem, Scheduling With Network Tasks, in which, along with traditional (CPU) tasks, network tasks have to be scheduled on network machines. We proposed two algorithms to solve the problem, G-List and Partition, for which we derive some theoretical guarantees. We demonstrated their effectiveness using datasets built using statistics from Google data center traces [13].
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.

Acknowledgment

This research was supported by the France 2030 program, managed by the French National Research Agency under grant agreements No. ANR-23-PECL-0003 and and ANR-22-PEFT-0002. It was also funded in part by the European Network of Excellence dAIEDGE under Grant Agreement Nr. 101120726, by SmartNet and LearnNet, and by the French government National Research Agency (ANR) through the 3IA Cote d'Azur Investments (ANR-23-IACL-0001), the UCA JEDI (ANR-15-IDEX-01) and EUR DS4H (ANR-17-EURE-004).

References

[1] F. Giroire, N. Huin, A. Tomassilli, and S. Perennes, "When Network Matters: Data Center Scheduling with Network Tasks," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. Paris, France: IEEE, Apr. 2019, pp. 2278–2286. [Online]. Available: https://ieeexplore.ieee.org/document/8737415/

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] 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, pp. 59–72.

[4] 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.

[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 *Proceedings*

of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012, pp. 2–2.

[6] 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. ACM, 2009, pp. 51–62.

[7] 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. ACM, 2008, pp. 75–86.

[8] 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. ACM, 2011, pp. 98–109.

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

[10] 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.

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

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

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

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

[15] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, and L. Li, "Towards practical and near-optimal coflow scheduling for data center networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3366–3380, 2016.

[16] Z. Li, Y. Shen, B. Yao, and M. Guo, "Ofscheduler: a dynamic network optimizer for mapreduce in heterogeneous cluster," *International Journal of Parallel Programming*, vol. 43, no. 3, pp. 472–488, 2015.

[17] F. Chen, M. Kodialam, and T. Lakshman, "Joint scheduling of processing and shuffle phases in mapreduce systems," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1143–1151.

[18] 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. ACM, 2015, pp. 407–420.

[19] 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.

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

[21] 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.

[22] V. J. Rayward-Smith, "Uet scheduling with unit interprocessor communication delays," *Discrete Applied Mathematics*, vol. 18, no. 1, pp. 55–71, 1987.

[23] 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.

[24] 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.

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

[26] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015, pp. 149–161.

[27] 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.

[28] R. G. Michael and S. J. David, "Computers and intractability: a guide to the theory of np-completeness," *WH Free. Co., San Fr*, pp. 90–91, 1979.

[29] G. Even, J. Naor, S. Rao, and B. Schieber, "Fast approximate graph partitioning algorithms," *SIAM Journal on Computing*, vol. 28, no. 6, pp. 2187–2214, 1999.

[30] 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.

[31] R. Krauthgamer, J. Naor, and R. Schwartz, "Partitioning graphs into balanced components," in *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 2009, pp. 942–949.

[32] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *The Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[33] Z. Liu and S. Cho, "Characterizing machines and workloads on a google cluster," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 2012, pp. 397–403.