# A Path Relinking Method for the Joint Online Scheduling and Capacity Allocation of DL Training Workloads in GPU as a Service Systems

Federica Filippini, Marco Lattuada, Michele Ciavotta, Arezoo Jahani, Danilo Ardagna, Edoardo Amaldi

**Abstract**—The Deep Learning (DL) paradigm gained remarkable popularity in recent years. DL models are used to tackle increasingly complex problems, making the training process require considerable computational power. The parallel computing capabilities offered by modern GPUs partially fulfill this need, but the high costs related to GPU as a Service solutions in the cloud call for efficient capacity planning and job scheduling algorithms to reduce operational costs via resource sharing. In this work, we jointly address the online capacity planning and job scheduling problems from the perspective of cloud end-users. We present a Mixed Integer Linear Programming (MILP) formulation, and a path relinking-based method aiming at optimizing operational costs by (i) rightsizing Virtual Machine (VM) capacity at each node, (ii) partitioning the set of GPUs among multiple concurrent jobs on the same VM, and (iii) determining a due-date-aware job schedule. An extensive experimental campaign attests the effectiveness of the proposed approach in practical scenarios: costs savings up to 97% are attained compared with first-principle methods based on, e.g., Earliest Deadline First, cost reductions up to 20% are obtained with respect to a previously proposed Hierarchical Method and up to 95% against a dynamic programming-based method from the literature. Scalability analyses show that systems with up to 100 nodes and 450 concurrent jobs can be managed in less than 7 seconds. The validation in a prototype cloud environment shows a deviation below 5% between real and predicted costs.

**Index Terms**—GPU as a Service, Scheduling, Capacity Allocation, Deep Learning training jobs.

## 1 Introduction

Nowadays, Deep Learning (DL) algorithms are used to tackle complex problems. These require to process massive datasets to train Neural Networks (NNs) with millions of parameters, which need to be tuned so as to achieve reasonable prediction accuracy and generality. Model compression and acceleration techniques [1] are employed to reduce the dimensionality of the training problems. However, they can only mitigate the inherent complexity of the learning task. The introduction of the General Purpose computation on Graphic Processing Units (GPGPU), providing an interface to massive parallelism, significantly extended the set of previously intractable problems that can be solved within a reasonable computing time. Consequently, the market growth for GPU as a Service systems is expected to be impetuous in the next years, starting from over 700 million USD in 2018 and increasing with a compound annual rate of over 38% up to 2025 [2]. GPU acceleration, especially the possibility of efficiently performing matrix multiplication in parallel thanks to highly specialized linear algebra libraries, is particularly suited to DL training tasks, providing about one order of magnitude reduction in the execution time with respect to CPU systems [3]. Moreover, an additional performance gain is ensured by the fact that Deep Neural Network models are often designed specifically to be deployed on GPU-based systems, taking full advantage of their architecture.

- *F. Filippini, M. Lattuada, D. Ardagna and E. Amaldi are with Politecnico di Milano. Milan, Italy*
  *Email: {name.lastname}@polimi.it*
- *M. Ciavotta is with Università di Milano-Bicocca, Milan, Italy*
  *Email: michele.ciavotta@unimib.it*
- *A. Jahani is with Sahand University of Technology, Tabriz, Iran*
  *Email: a.jahani@sut.ac.ir*

Despite the achieved progress, training DL models remains a computationally intensive task. Furthermore, high performance servers with support to GPU-accelerated applications are characterized by considerably high costs (about 200k USD for high-end systems like NVIDIA DGX A100 [4]). Consequently, they are often unaffordable for the general public, which includes small organizations. This growing demand and the issues related to the accessibility of GPU-based architectures caused, in the last years, a progression of cloud services aiming at democratizing the access of those resources in different contexts with pay-as-you-go pricing models.

Requiring no upfront investments and infrastructure maintenance expenditures, GPU-enabled solutions became accessible to a wider range of organizations. However, the time-unit cost of GPU-based Virtual Machines (VMs) is still remarkably high, being 5-8x more expensive than those of VMs featuring only CPUs [5]. As a consequence, selecting the most suitable resources to co-locate different DL training workloads and determining efficient schedules is crucial for an effective implementation of the *GPU as a Service* model.

To the best of our knowledge, most of the existing literature focuses on either the scheduling or the resource selection aspect, relying on users' requests in terms of resources (i.e., specifying the type and number of GPUs) to assign to the submitted workloads (see, e.g., [6], [7]) or delegating the job scheduling to simple mechanisms (e.g., First In First Out; see, for instance, [8], [9]), respectively. In this work, we tackle the online resource selection and job scheduling problems jointly and from a higher level perspective, where users are only asked to specify: (i) a due date for each DL training job, (ii) a priority associated with the fulfillment of the due date. The objective is to minimize the usage cost of cloud VMs and the tardiness costs (i.e., penalty costs related to the difference between jobs completion times and their due dates). We envision a reference scenario where multiple workloads are continuously submitted for execution on a dynamic cluster of VMs. Individual nodes can be dynamically configured from various VM types available in the cloud provider's catalog, each one featuring possibly several GPUs. A single node can host more than one job, and, in this case, the available resources are partitioned and statically allocated to avoid interference. The set of jobs to be scheduled is not known in advance: new jobs are submitted with different characteristics, due dates, and tardiness penalties without any repetition scheme, leading to an online problem. Finally, job preemption is allowed to manage higher priority submissions.

Online decisions concern selecting the VM type for each node, the order in which the jobs are executed, and how resources are partitioned and assigned to each job. We formalize the optimization problem arising at each decision point in time through a Mixed Integer Linear Programming (MILP) formulation, which however is too challenging to be solved directly in practical scenarios. Therefore, we develop a heuristic algorithm based on randomized greedy and path relinking [10]. This method is specifically designed and implemented to be efficient while achieving high-quality solutions. Our experimental campaign demonstrates its effectiveness in practical scenarios, since solutions for systems with up to 100 nodes and 450 concurrent jobs are obtained in less than 7 seconds. Significant costs savings are attained by our path relinking-based algorithm with respect to first-principle methods (based on Earliest Deadline First, First In First Out, Priority Scheduling), with an average percentage gain between 23 and 97%. We obtain an average cost reduction between 7 and 20% with respect to the Hierarchical Method we developed in our previous work [11] as a first approach to address the complexity of the problem. Finally, we reduce the costs between 43 and 95% compared to a dynamic programming (DP)-based method proposed in the literature [9] purposely modified to adapt to our problem. The validation of our approach in a cloud prototype environment based on Microsoft Azure showed a deviation below 5% between real and predicted costs.

The rest of the paper is organized as follows. Section 2 describes the reference framework for our work, focusing on the characterization of the system architecture and of the problem, and presenting the Machine Learning models used to predict training jobs performance. Section 3 describes the MILP formulation and the developed heuristic method. Section 4 illustrates the experimental setup and compares the results of the proposed algorithm with those obtained with first-principle methods, our previous Hierarchical Method [11] and the DP-based method [9]. Section 5 relates our work to the literature. Conclusions are finally drawn in Section 6.

## 2 Cloud Framework and Problem Description

In this paper, we address the problem of scheduling DL training jobs on GPU as a Service systems from the point of view of the cloud end-users. We consider a complex system involving multiple nodes that can be provisioned on-demand in the cloud and configured with VMs of different types, each one with a possibly different number of GPUs. The submitted jobs can be executed concurrently on the assigned nodes; all computational resources are shared except for GPUs, that are dedicated to running single jobs. The considered problem and the reference cloud system are presented in Section 2.1.

As a prerequisite to successfully address our problem (and, in particular, to rightsize the type and amount of resources to be assigned to the jobs), we must estimate the training time of the DL applications reliably. This prediction is performed via the Machine Learning (ML) models presented in Section 2.2.

### 2.1 System Architecture and Problem Statement

The problem we address encompasses three intertwined subproblems: (i) a job scheduling problem that consists in determining which jobs to run among those available and assigning them to the available nodes; (ii) a capacity allocation problem that consists in selecting the most appropriate number of nodes and the best VM type for each node; (iii) a resource partitioning problem, that consists, for each node, in partitioning the available GPUs among the selected jobs.

This joint problem is solved in an online setting, i.e., every time a new job is submitted or a running job completes. Nevertheless, if none of these situations occurs, the system is reconfigured after a fixed time interval, denoted in the following by $H$. The value of $H$ is conventionally set to one hour since the cloud pricing models usually consider the hour as the unit of billing [12], [13], while the average time required by a DL training job is in the order of tens to hundreds of hours. As in other literature proposals, we allow migration [7], [14] and preemption [14], [15] of running jobs; therefore, all the previously made decisions can be modified at any rescheduling point. Specifically, running jobs can either continue their execution with a possibly different configuration (in terms of VM type and number of GPUs) or be preempted and restarted from the last checkpoint in a future step. Saving periodic snapshots of the model is common practice in DL operations to increase fault tolerance, which comes at the cost of some overhead. In this study, however, the checkpointing time overhead is considered included in the job execution time because: (i) the former is often negligible when compared to the overall job duration, and (ii) the training time prediction models presented in Section 2.2 are agnostic of the operations performed by the application during the training process.

The online nature makes our problem even more complex to tackle since, at each rescheduling point, no information is available on the number and characteristics of jobs that will be submitted in the future. In particular, the unknown job interarrival time and resource requirements distribution make it challenging to identify an effective strategy to minimize the overall schedule expected cost, like in zero-data problems [16], where the system cannot effectively characterize in advance the incoming workload profile. Indeed, the ultimate goal is to optimize the long-term execution cost in scenarios where multiple jobs are submitted and executed at different points over a long time horizon considering resource leasing costs and due date violation penalties. In this context, the choices made at each scheduling point can have unforeseen repercussions on the system status at the next decision point due to the partial execution of some applications. Thus, locally optimal choices, which guarantee the lowest possible execution cost in the current period, may lead to a worse outcome when the overall schedule is considered.

Formally, our reference cloud system includes a set $\mathcal{N} = \{n_1, n_2, ..., n_N\}$ of processing units referred to as nodes, such that $N = |\mathcal{N}|$. Each node is a VM that can be configured on-demand with a type selected from the cloud provider's catalog,

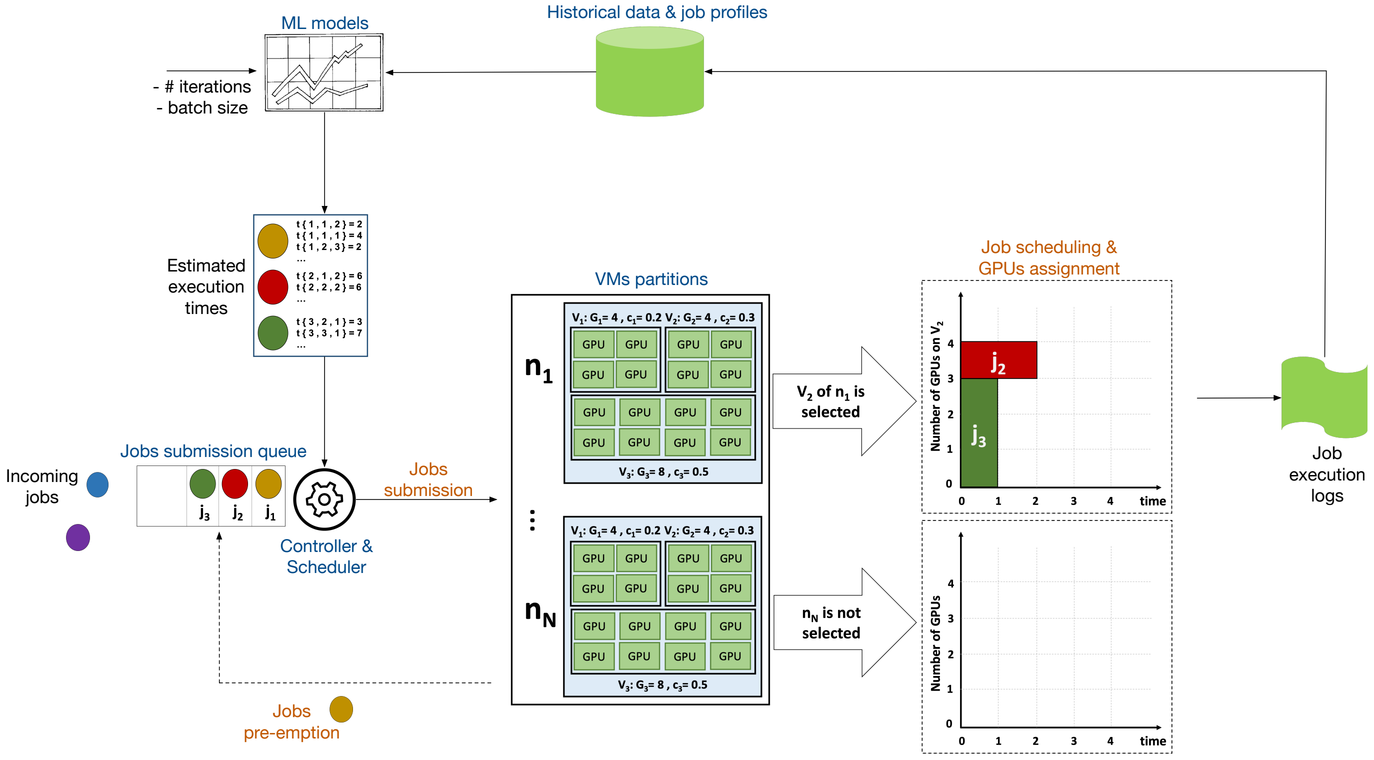


Figure 1: Reference cloud framework. In the example, we consider three submitted jobs $j_1, j_2, j_3$. The values of execution times $t_{jvg}$ are chosen by way of example. Nodes $n_1$ to $n_N$ can be equipped with three types of VMs: $v_1$ with 4 GPUs and cost $c_1 = 0.2\$/h$, $v_2$ with 4 GPUs and $c_2 = 0.3\$/h$, and $v_3$ with 8 GPUs and $c_3 = 0.5\$/h$. Jobs $j_2$ and $j_3$ are deployed on $n_1$ with VM $v_2$. In particular, $j_2$ runs on 1 GPU, while $j_3$ on 3 GPUs. Job $j_1$ is sent back to the queue and no other nodes are selected.

denoted by the set $\mathcal{V}$. Each VM type $v \in \mathcal{V}$ is characterized by a number $G_v$ of available GPUs and by a time-unit cost $c_v$. The set of jobs available at a given time instant, including the jobs in execution and in the waiting queue, is denoted by $\mathcal{J}$. Each job is characterized by its release time, its due date $d_j$, and by an estimated processing time $t_{jvg}$ (to perform a fixed number of training iterations) that depends on the VM type $v$ and the number $g$ of GPUs assigned to it, as described in Section 2.2. The batch size and the number of iterations performed by the jobs are considered as fixed and known. Jobs are never rejected, but, as mentioned, they can be preempted and postponed if a job with higher priority is submitted and the available resources are insufficient to process all of them. The priority of a job depends on the residual processing time, the due date, and the associated tardiness penalty. The tardiness of job $j \in \mathcal{J}$ is denoted by $\tau_j$, and a weight $\omega_j$ is used to compute the tardiness cost, $\omega_j \tau_j$.

The scheduling process is summarized in Figure 1. At any time a new job $j$ is submitted to the system or a running job is completed (referred to as rescheduling point), the list of incomplete jobs is virtually merged[1] with the queue of unstarted jobs, sorted by submission time, giving rise to the queue $\mathcal{J}$ that will be the input of our scheduler. Then, the estimated execution time of partially executed jobs is updated, and the problem of determining the nodes to be used and the best available configuration (VM type) to be assigned to each selected node is solved. As in other literature proposals [14], [17]–[19], we assume that multiple jobs can be deployed on the same node, while, within each node, each job receives for exclusive use a certain number of GPUs. As will be observed in Section 4.6, the interference experienced among jobs in the same VM is negligible in our setting. This does not hold, however, when co-locating multiple jobs on the same GPU, since the contentions significantly affect the runtime performance. Therefore, we decided to consider GPU sharing as part of our future work.

The jobs not selected for execution are kept in the queue until the following rescheduling point. The job selection and resource allocation processes aim at executing all jobs in $\mathcal{J}$ before their due date and, simultaneously, at maximizing the utilization of the selected nodes avoiding idle resources, thus reducing the overall cost. The latter is given by the sum of the execution costs of all jobs and of the tardiness cost of jobs whose due date is violated.

The ML models used to predict job execution times, which are described in the next section, are originally trained using historical data and information coming from pre-profiling. Execution logs of all the running jobs are collected and can be used to periodically re-train the models, enhancing their accuracy.

1. For the sake of simplicity, we consider in our method a unique virtual queue $\mathcal{J}$. In practice, jobs that are already in execution are possibly stopped and reallocated only at the end of the scheduling process, if the type or the amount of resources they receive should be modified.

## 2.2 ML Models for Predicting Training Jobs Performance

To solve the scheduling and resource allocation problem described in the previous section, it is paramount to lean on reliable estimates of the execution time for each submitted job on each available hardware configuration (i.e., on each available combination of number and type of GPUs). The duration of individual DL jobs depends on many factors, some of which are deterministic, such as the type of the implemented architecture, the batch size, the number of iterations, and the stopping policy. Others, instead, have a random nature (the initial distribution of the weights) or are hardly knowable a priori (e.g., the characteristics of the training dataset). Thus, it is not possible to know the job duration with certainty, and performance estimation techniques are needed. Since the considered problem must be solved online and in a reduced amount of time, simulation-based methods are not feasible.

In this work, ML models are adopted to correlate job and target VM features with the expected execution time. In particular, as discussed in [20], linear regression can be used to automatically build models that infer a deep network training time on a particular type of VM. A different prediction model is built in [20] for each type of neural network. The considered features related to the characteristics of the training jobs are the number of iterations and the batch size. Vice versa, the considered features of the VMs are the nominal computational power of the GPUs (in terms of GFLOPS), the number of GPUs, and the performance of the disk (i.e., the time in seconds to load 120k files of 192 kB from disk into the memory of a GPU). These numerical features and their inverses are combined to build an extended set that is filtered through sequential forward feature selection [21] and used to feed the ML models. It is worth noting that, since the models consider the number of iterations of the training process, they can be used not only to estimate the execution time for the full execution of the job, but also to predict the time required to complete the remaining iterations of an already started job, even if it is migrated to a different VM type [20], [22].

As reported in [22], the worst-case mean absolute percentage error is below 11%, making the models accurate enough to be exploited in the considered scenario (see also Section 4.6).

## 3 Problem Formulation and Solution

This section aims at presenting the optimization model and the heuristic method we developed to tackle this joint Job Scheduling (JS) and Capacity Allocation (CA) problem.

In Section 3.1 we describe our mathematical optimization formulation, which accurately models the cost structure of the problem arising at every rescheduling point. Its complexity, reflected by the very large number of variables and constraints, calls for designing efficient heuristic algorithms to determine good-quality solutions in reasonable computing time. Since at each rescheduling point the optimization process may return decisions that are myopically optimized for the current context, in Section 3.2 we study an alternative objective function with the goal of improving the results in the long run. Finally, in Section 3.3 we propose a heuristic method for this problem, which integrates a randomized greedy procedure and the path relinking strategy [10].

### 3.1 Optimization Model

Due to the online nature of the problem, the formulation presented in this work refers to the *local* optimization problem, as defined in Section 2.1, where the decisions made at each scheduling point only take effect until the next event occurs. This approach has significant consequences, primarily on the local objective function used to guide the optimization process toward a good quality *global* solution. For this reason, the Mixed Integer Linear Programming (MILP) formulation presented in this section is focused on the computation of the deployment cost of the first-ending job. This is well-suited to model the behavior of the online scheduling problem: the system is reconfigured not only every time a new job is submitted, but also every time a job is completed. Jobs completions are the only predictable events in our online setting, and all resources might be reallocated after a rescheduling; therefore, it is reasonable to bind the overall cost of the schedule to the deployment cost of the first-ending job on each node.

Table 1: Problem parameters and variables

| **Parameters** | |
|---|---|
| $\mathcal{J}$ | set of submitted jobs |
| $\mathcal{N}$ | set of nodes |
| $\mathcal{V}$ | set of VM types |
| $G_v$ | number of available GPUs on a VM of type $v \in \mathcal{V}$ |
| $\mathcal{G}_v$ | set of available GPUs on a VM of type $v \in \mathcal{V}$; $\mathcal{G}_v = \{1, 2, ..., G_v\}$ |
| $c_v$ | time unit cost of a VM of type $v \in \mathcal{V}$ |
| $d_j$ | due date of job $j \in \mathcal{J}$ |
| $\omega_j$ | tardiness weight of job $j \in \mathcal{J}$ |
| $M_j^t$ | maximum execution time of job $j \in \mathcal{J}$ |
| $t_{jvg}$ | execution time of job $j \in \mathcal{J}$ on the VM type $v \in \mathcal{V}$ with $g$ GPUs |
| $\widehat{M_j^c}$ | maximum possible deployment cost for job $j \in \mathcal{J}$ |
| $H$ | scheduling time interval |
| $\mu$ | penalty coefficient for idle GPUs |
| $\rho$ | penalty coefficient for postponed jobs |
| **Variables** | |
| $w_n$ | 1 if node $n \in \mathcal{N}$ is selected, 0 otherwise |
| $y_{nv}$ | 1 if the VM type $v \in \mathcal{V}$ is selected on node $n \in \mathcal{N}$, 0 otherwise |
| $z_{jn}$ | 1 if job $j \in \mathcal{J}$ is executed on node $n \in \mathcal{N}$, 0 otherwise |
| $x_{jnvg}$ | 1 if job $j \in \mathcal{J}$ is executed on node $n \in \mathcal{N}$ with a VM of type $v \in \mathcal{V}$ and $g$ GPUs, 0 otherwise |
| $\tau_j$ | tardiness of job $j \in \mathcal{J}$ |
| $\hat{\tau}_j$ | worst-case tardiness of job $j \in \mathcal{J}$ when it is postponed |
| $\pi_{jn}$ | deployment cost of job $j \in \mathcal{J}$ on node $n \in \mathcal{N}$ |
| $\alpha_{jn}$ | 1 if job $j \in \mathcal{J}$ is the first-ending job on node $n \in \mathcal{N}$, 0 otherwise |

As already mentioned, we consider four input sets: the set of candidate jobs $\mathcal{J}$, the set of nodes $\mathcal{N}$, the set of VM types $\mathcal{V}$ (according to current cloud providers' catalogues), and the set of GPU partitions for each VM type $v$, denoted by $\mathcal{G}_v$. In particular, assuming homogeneous GPUs for all $v \in \mathcal{V}$, $\mathcal{G}_v = \{1, \dots, G_v\}$, where $G_v$ is the total number of GPUs available on VM type $v$. The constant parameter $H$ represents the periodic scheduling time interval. The problem parameters and variables are summarized in Table 1.

The proposed MILP formulation for the joint JS and CA problem is as follows:

$$\min \sum_{j\in\mathcal{J}} (\omega_j \tau_j + \rho\omega_j\hat{\tau}_j) + \mu \sum_{\substack{n\in\mathcal{N}\\ v\in\mathcal{V}}} \left( G_v y_{nv} - \sum_{\substack{j\in\mathcal{J}\\ g\in\mathcal{G}_v}} g x_{jnvg} \right) + \sum_{\substack{j\in\mathcal{J}\\ n\in\mathcal{N}}} \pi_{jn}\alpha_{jn} \tag{P1a}$$

subject to:

$$\sum_{v\in\mathcal{V}} y_{nv} = w_n \qquad \forall n \in \mathcal{N} \tag{P1b}$$

$$x_{jnvg} \le y_{nv} \qquad \forall j \in \mathcal{J}, \forall n \in \mathcal{N}, \forall v \in \mathcal{V}, \forall g \in \mathcal{G}_v \tag{P1c}$$

$$x_{jnvg} \le z_{jn} \qquad \forall j \in \mathcal{J}, \forall n \in \mathcal{N}, \ \forall v \in \mathcal{V}, \ \forall g \in \mathcal{G}_v \tag{P1d}$$

$$\sum_{v\in\mathcal{V}, g\in\mathcal{G}_v} x_{jnvg} \le w_n \qquad \forall j \in \mathcal{J}, \ \forall n \in \mathcal{N} \tag{P1e}$$

$$\sum_{n\in\mathcal{N}, v\in\mathcal{V}, g\in\mathcal{G}_v} x_{jnvg} = \sum_{n\in\mathcal{N}} z_{jn} \qquad \forall j \in \mathcal{J} \tag{P1f}$$

$$\sum_{j\in\mathcal{J},g\in\mathcal{G}_v} g x_{jnvg} \le G_v \qquad \forall n\in\mathcal{N},\ \forall v\in\mathcal{V} \tag{P1g}$$

$$\sum_{n\in\mathcal{N},v\in\mathcal{V},g\in\mathcal{G}_v} t_{jvg} x_{jnvg} \le d_j+\tau_j \qquad \forall j\in\mathcal{J} \tag{P1h}$$

$$\left(H+M_j^t\right)\left(1-\sum_{n\in\mathcal{N}} z_{jn}\right) \le d_j+\hat{\tau}_j \qquad \forall j\in\mathcal{J} \tag{P1i}$$

$$\sum_{v\in\mathcal{V},g\in\mathcal{G}_v} t_{jvg} c_v x_{jnvg} \le \pi_{jn} \qquad \forall j\in\mathcal{J},\ \forall n\in\mathcal{N} \tag{P1j}$$

$$\sum_{j\in\mathcal{J}} \alpha_{jn} = w_n \qquad \forall n\in\mathcal{N} \tag{P1k}$$

$$\alpha_{jn} \le z_{jn} \qquad \forall j\in\mathcal{J},\ \forall n\in\mathcal{N} \tag{P1l}$$

$$\sum_{n\in\mathcal{N}} z_{jn} \le 1 \qquad \forall j\in\mathcal{J} \tag{P1m}$$

$$\sum_{n\in\mathcal{N}} w_n = \min\{N,J\} \tag{P1n}$$

$$w_n\in\{0,1\} \qquad \forall n\in\mathcal{N} \tag{P1o}$$

$$y_{nv}\in\{0,1\} \qquad \forall n\in\mathcal{N},\ \forall v\in\mathcal{V} \tag{P1p}$$

$$z_{jn}\in\{0,1\},\ \ \pi_{jn}\ge 0,\ \ \alpha_{jn}\in\{0,1\} \qquad \forall j\in\mathcal{J},\ \forall n\in\mathcal{N} \tag{P1q}$$

$$\tau_j\ge 0,\ \ \hat{\tau}_j\ge 0 \qquad \forall j\in\mathcal{J} \tag{P1r}$$

$$x_{jnvg}\in\{0,1\} \qquad \forall j\in\mathcal{J},\ \forall n\in\mathcal{N},\ \forall v\in\mathcal{V},\ \forall g\in\mathcal{G}_v \tag{P1s}$$

Constraints (P1b) enforce that exactly one VM type $v\in\mathcal{V}$ is selected for each open node $n\in\mathcal{N}$. Notice that, here and in the following, we denote a node $n\in\mathcal{N}$ as *open* if $w_n=1$, i.e., the node has been selected. Since $y_{nv}=1$ only if a VM of type $v$ is deployed on node $n$, $w_n=0$ enforces $y_{nv}=0$ for all $v\in\mathcal{V}$. Conversely, when $w_n=1$, $\sum_{v\in\mathcal{V}} y_{nv}$ can be 1 only if exactly one VM type $v$ is chosen on node $n$.

Constraints (P1c), (P1d), (P1e) and (P1f) prescribe that each job $j\in\mathcal{J}$ can be assigned only to configurations (i.e., nodes, VMs and number of GPUs) that have actually been selected. In particular, according to inequalities (P1c), $x_{jnvg}$ can be equal to 1, i.e., job $j$ can be executed on node $n$ with VM type $v$ and $g$ GPUs, only if $y_{nv}=1$, that is, node $n$ is open and equipped with a VM of type $v$. Constraints (P1d) state, instead, that each job $j$ can be deployed with any configuration $(v,g)$ on node $n$ only if it has been assigned to node $n$ in the current time period, i.e., $z_{jn}=1$. The sum at left-hand-side of Constraints (P1e) represents the total number of configurations $(v,g)$ selected for each job $j$ on node $n$. In particular, this must be 0 when node $n$ is closed, i.e., when $w_n=0$. Conversely, when $w_n=1$ at most one of the variables $x_{jnvg}$ can be equal to 1, i.e., one single configuration $(v,g)$ must be assigned to job $j$ if it is deployed on node $n$, while all $x_{jnvg}$ are 0 if job $j$ is postponed or it is executed on a different node. Moreover, according to Constraints (P1f), if a job $j$ is selected to run on node $n$, i.e., $z_{jn}=1$, exactly one $x_{jnvg}$ must be 1, that is, the job must be deployed on one node and assigned to a single configuration $(v,g)$.

Constraints (P1g) enforce that the total number of GPUs assigned to the jobs deployed on a given node $n$ does not exceed the number of available GPUs on the VM type $v$ selected on the node. Since $x_{jnvg}=1$ only if job $j$ is executed on node $n$ with exactly $g$ GPUs, the left-hand-side of Constraints (P1g) is equal to the total number of GPUs assigned to jobs deployed on node $n$ and VM $v$.

Constraints (P1h) and (P1i) are used to define the tardiness $\tau_j$ and worst-case tardiness $\hat{\tau}_j$ of all jobs $j\in\mathcal{J}$. In particular, the total execution time of job $j\in\mathcal{J}$ on the selected configuration is bounded by the sum of its due date and its tardiness if the job is executed (see Constraints (P1h)). If, in turn, the job is postponed, i.e., the sum of $z_{jn}$ over all nodes $n\in\mathcal{N}$ is equal to zero, the worst-case tardiness $\hat{\tau}_j$ is defined by inequality (P1i) to be greater than or equal to the sum between the maximum execution time of job $j$ and the scheduling interval $H$, minus the due date $d_j$.

Constraints (P1j) enforce the deployment cost of job $j$ on node $n$, denoted by $\pi_{jn}$, to be greater than or equal to the execution cost of the job itself, expressed by the product between the execution time on the selected configuration and the cost of the selected VM type. Notice that, since the variables $\pi_{jn}$ are multiplied by $\alpha_{jn}\ge 0$ in the objective function to be minimized, this set of constraints defines the deployment costs. Constraints (P1k) and (P1l) state that, for each open node $n$, exactly one job is selected among those that complete first on $n$, while Constraints (P1m) prescribe that each job must be assigned to at most one node. This locality constraint is introduced in our model for the sake of simplicity. Moreover, the analysis of production systems traces reported in [23] and discussed in [14] highlights how almost 87% of the jobs require a single GPU, and that the degree of interference is higher for jobs that are distributed on many different servers. It is observed that accessing GPUs from multiple VMs may introduce significant communication overheads in general, and it is therefore considered as an antipattern when the overall number of required GPUs is small enough to fit in a single node. In particular, these considerations are considered in [14] as incentives to exploit jobs migration and packing to improve resource utilization. More recently, [24] points out that 85% of the jobs can be deployed on single servers hosting up to 8 GPUs, as in the scenarios we considered in the experiments (see Section 4.1). However, as the DL models are growing fast, doubling their size every 3.4 months [25], distributed training is unavoidable for some applications. The extension of model (P1) to situations where multiple servers can be used to deploy a job is left as part of our future work.

Having denoted with $J$ and $N$ the cardinality of $\mathcal{J}$ and $\mathcal{N}$, respectively, Constraints (P1n) require that a suitable number of nodes is used, so that as many submitted jobs as possible are executed. Since jobs that are postponed become closer to their due date and, therefore, require more (and possibly more expensive) resources to be completed without tardiness, it is reasonable to run all jobs as soon as resources are available to reduce the overall execution costs. In an online framework, with unpredictable arrival times and new jobs characteristics, the resource assignment at a given time may affect the effectiveness of the solution in the following periods. However, preemption mitigates this issue by allowing, at any decision point, to revise the resource assignment considering newcomer jobs. Finally, Constraints (P1o)-(P1s) define the decision variables domain.

The objective function (P1a) includes three terms. The first represents the sum over all jobs $j\in\mathcal{J}$ of the tardiness cost and the worst-case tardiness cost. The latter occurs when the job is postponed and is used to penalize delaying a job. Although, when considering the long-term scenario, postponed jobs may still be completed within the due date, the local decision-maker cannot foresee the future system state and aims at identifying a robust solution considering the worst-case scenario. Both $\tau_j$ and $\hat{\tau}_j$ are multiplied by the tardiness weight $\omega_j$ that denotes the priority of each job: violating the due date of high-priority jobs implies a higher penalty. Moreover, $\hat{\tau}_j$ is multiplied by a coefficient $\rho > 1$, which further penalizes jobs postponements.

The second term in the objective function (P1a) represents the difference between the number of assigned GPUs and the number of available GPUs on all nodes. Thus, through the positive constant $\mu$, it penalizes solutions with idle resources. In particular, for each node $n$, only the variable $y_{nv}$ corresponding to the selected VM type is equal to 1. Therefore, $\sum_{n,v} G_v y_{nv}$ is the total number of available GPUs on the chosen nodes. The second term $\sum_{j,g} g x_{jnvg}$ is equal, as in Constraints (P1g), to the number of GPUs assigned to jobs deployed on node $n$. Thus, the difference gives the total number of idle GPUs.

The third term corresponds to the total execution cost: for each node $n \in \mathcal{N}$, the variable $\alpha_{jn}$ is 1 if job $j$ is the first-ending job on $n$, according to the assigned configuration. The execution cost on node $n$ is computed as the deployment cost of the first-ending job on that node. This is reasonable since, as soon as a job is completed, the system is fully reconfigured, thus the resources assigned to all jobs may change. Note that we neglect the reconfiguration costs of the running nodes, as well as the costs due to preemption overheads, since both events are orders of magnitude faster than DL training time(few minutes against several hours or days). Due to the last term of the objective function, the resulting optimization model (P1) is nonlinear. The linearization is available as Appendix A in the additional material.

As a final consideration, the huge number of variables and constraints makes Problem (P1) beyond the reach of the state-of-the-art MILP solvers, even for small-size instances of the problem. While the mathematical formulation is crucial to formalize the problem, heuristic methods are required to determine good-quality solutions within a reasonable computing time. As mentioned before, an attempt to exploit the MILP formulation had been made in [11], where the set of jobs $\mathcal{J}$ and the set of nodes $\mathcal{N}$ were partitioned and managed by a set $\mathcal{K}$ of local controllers. Jobs were assigned to controllers according to a round robin policy while each local controller was in charge of solving very small instances (with $|\mathcal{N}| \leq 5$) of a MILP problem equivalent to (P1) using a state-of-the-art solver. This divide-et-impera approach, despite delivering some good-quality solutions in a reasonable computing time, explores only a limited portion of the solution space. Consequently, the optimality gap of its solutions is generally wide and grows very steeply with $J$. In this work, we consider the problem in a centralized framework and we tackle it through the heuristic algorithm proposed in the next sections. The experimental results reported in the following sections prove how a centralized, albeit heuristic, method outperforms significantly the previous approach [11], both in terms of solution quality and scalability.

### 3.2 Alternative Proxy Function

As mentioned in Section 2.1, our ultimate goal is to optimize the costs of a long-term scenario, including multiple job submissions and executions. The total costs of the candidate solutions obtained with our heuristic algorithm, described in the following section, are evaluated through the function $f_{\text{OBJ}}$, that implements the objective function reported in Equation (P1a). Since this function is also exploited to make local scheduling decisions that over time build up the global schedule, we often refer to $f_{\text{OBJ}}$ as a *proxy* function. This baseline proxy function focuses on cost minimization during the current scheduling step. This approach provides high-quality local solutions but proves to be short-sighted in terms of the long-term scenario. Indeed, being conservative in the use of resources (favoring the cheapest available configuration), it runs the risk of increasing the pressure of jobs waiting to be processed to the point of having to compensate for them through costly configurations.

In order to partially address this issue, we developed a different proxy function to be used in place of $f_{\text{OBJ}}$, that will be denoted in the following as $\bar{f}_{\text{OBJ}}$:

$$\bar{f}_{\text{OBJ}} = \max \sum_{j \in \mathcal{J}} \frac{M_j^t}{\pi_j + \omega_j \tau_j}. \tag{1}$$

It aims at favoring the schedules that use the resources most efficiently, pursuing a trade-off between maximizing the number of completed jobs and minimizing the operational costs. The maximum processing time $M_j^t$ at the numerator is used as a scaling factor to weight the importance of all jobs in the queue, favoring long-running jobs. The denominator is composed of two terms: the deployment cost $\pi_j$ of each job and the penalty cost $\omega_j \tau_j$ associated with the due date violation.

Such definition shifts the focus to resource efficiency: each term decreases as the processing time and the tardiness increase. Consequently, the model tries to keep these values small for all jobs. As we shall see in Section 4.3, $\bar{f}_{\text{OBJ}}$ provides better experimental results when applied in the centralized heuristic scheme, based on path relinking, described in the next section.

### 3.3 Proposed Algorithmic Solution

We developed a heuristic algorithm to swiftly identify good-quality solutions for the optimization problem presented in Section 3.1, combining and adapting randomized greedy and path relinking schemes. The proposed method is based on the following assumptions:

- Jobs are sorted by their pressure $\Delta_j$, which measures how close they are to the due date when executed with the fastest configuration. The pressure is defined as:

$$\Delta_j = T_c + \min_{v \in \mathcal{V}, g \in \mathcal{G}_v} \{t_{jvg}\} - d_j, \tag{2}$$

  where $T_c$ denotes the current rescheduling point.
- The optimal configuration $(v, g) \in \mathcal{V} \times \mathcal{G}_v$ for each selected job is either (i) the cheapest configuration such that the job is executed before its due date, if such a configuration exists, or (ii) the fastest available configuration if, independently from the selected setup, it is not possible to execute the job before its due date.
  In our framework, the time-unit deployment cost of a job on any available configuration is always lower than the penalty incurred if the job due date is violated. Therefore, such assumption allows to minimize costs also when due dates are violated, completing the job execution as fast as possible to reduce the corresponding penalty.
- Deployment costs increase linearly with the number of GPUs (see the cloud providers pricing models [12], [13]).

- Processing times speed-up is sublinear in the number of GPUs (as observed in, e.g., [22]).

The proposed method includes three main steps. In the *preprocessing* step, the pressure of all candidate jobs is computed as defined in Equation (2). Then, the *optimization* step implements a randomized construction procedure where the best available configuration is selected for all jobs in the pressure-sorted virtual queue $\mathcal{J}$, followed by a step of path relinking. Finally, the *postprocessing* step aims at reducing the amount of idle resources. Specifically, if a VM $v$ with $g_v$ available GPUs is selected on a node $n$, but only a fraction $g < g_v$ of them is used, we try to substitute it with a $v' \in \mathcal{V}$ that hosts GPUs of the same type and whose number is such that $g \leq g_{v'} < g_v$. Moreover, if this update cannot be performed or if, after having completed it, there are still nodes with idle GPUs, the idle resources on node $n$ are allocated to the job with the highest speed-up among those deployed on $n$.

We empirically evaluated the impact of the two components of the optimization step by means of an ablation study (presented in Section 4) in which we elicited variants considering the complete scheme (called *Path Relinking*), the method obtained by disabling the path relinking procedure (called *Randomized Greedy*) and finally the greedy algorithm obtained by removing also the randomization (*Greedy*). In a context where scalability is crucial to manage large-scale systems, it is important to highlight the benefits of each approach, since more complex algorithms require longer execution times. The costs of the scheduling decisions made by the algorithm are evaluated using a proxy function $f_P$. This corresponds to the objective function $f_{\text{OBJ}}$ of Equation (P1a) in the case of the pure Greedy or the Randomized Greedy method, and to the function $\bar{f}_{\text{OBJ}}$ defined in Equation (1) in the complete algorithm including the path relinking strategy. The overall costs of the proposed solutions are finally computed exploiting $f_{\text{OBJ}}$ (Equation (P1a)) at the end of a long-term scenario involving multiple submissions and the complete execution of all jobs. This allows us to compare also the results obtained with methods exploiting different proxy functions.

The optimization step is described in details in the following.

### 3.3.1 Optimization Step

The optimization step consists of two substeps that are executed sequentially. It starts with a randomized greedy construction procedure that is used to build a set $\mathcal{S}^*$ of good-quality candidate solutions, obtained by optimizing the proxy function $f_P$. This set is taken as a starting point for a path relinking procedure that improves the best candidate solution by iteratively identifying and combining features of the other solutions in $\mathcal{S}^*$. In the following, these procedures are presented in detail.

#### Randomized Construction Procedure

The randomized construction procedure is reported in Algorithm 1. At each iteration, a new candidate solution $S$ is built through a randomized greedy method. If it has a better $f_P$ value than any other solution currently stored in $\mathcal{S}^*$, the set is updated, possibly removing the worst-valued solution to keep its cardinality under a fixed value $\sigma$. Each candidate solution $S \in \mathcal{S}^*$ represents a data structure carrying information about the running jobs and the relative configuration. Specifically, each element of $S$ is a tuple $(j, n, v, g)$, where $j$ is the current job and $n$, $v$, $g$ are the node, VM type, and number of GPUs assigned to it, respectively.

**Algorithm 1** Randomized construction procedure

```
1:  function RANDOMIZED_CONSTRUCTION($\mathcal{J}$, MaxIt$_{\text{RG}}$)
2:      iter = 0
3:      $\mathcal{S}^* \leftarrow \emptyset$
4:      while iter < MaxIt$_{\text{RG}}$ do        ▷ MaxIt$_{\text{RG}}$: maximum number of random iterations
5:          $S \leftarrow$ empty schedule                ▷ $S$: current schedule
6:          $\mathcal{J}_s \leftarrow$ SORT_JOBS_LIST($\mathcal{J}$, $\Delta$)    ▷ $\Delta$ : pressures of all jobs
7:          for all $j \in \mathcal{J}_s$ do                ▷ $\mathcal{J}_s$ : sorted queue
8:              $D_j^* = \{(v,g)$ s.t. $t_{jvg} + T_c < d_j\}$
9:              $(v^*, g^*) \leftarrow$ SELECT_BEST_CONFIGURATION($j, D_j^*$)
10:             assigned $\leftarrow$ ASSIGN_TO_EXISTING_NODE($j, (v^*, g^*), \mathcal{N}_O$)
11:             if not assigned then
12:                 if $|\mathcal{N}_O| < N$ then
13:                     Select $\nu'$ with VM type $v^*$ and $G_{v^*}$ GPUs
14:                     $\mathcal{N}_O \leftarrow \mathcal{N}_O \cup \{\nu'\}$
15:                     $S \leftarrow S \cup (j, \nu', v^*, g^*)$
16:                 else
17:                     ASSIGN_TO_SUBOPTIMAL($j, S, \mathcal{N}_O$)
18:                 end if
19:             else
20:                 $S \leftarrow S \cup (j, \nu^*, v^*, g^*)$
21:             end if
22:             $\mathcal{J}_s \leftarrow \mathcal{J}_s \setminus \{j\}$
23:         end for
24:         if $f_P(S)$ is better than $f_P(S')$ for any $S' \in \mathcal{S}^*$ then
25:             $\mathcal{S}^* \leftarrow \mathcal{S}^* \cup \{S\}$
26:             $\mathcal{S}^* \leftarrow \mathcal{S}^* \setminus \{S'\}$ if $|\mathcal{S}^*| > \sigma$
27:         end if
28:         iter $\leftarrow$ iter + 1
29:     end while
30:     return $\mathcal{S}^*$
31: end function
```

Algorithm 1 proceeds as follows: at line 6, the set $\mathcal{J}$ of submitted jobs is sorted producing as output a new set denoted by $\mathcal{J}_s$. Note that job $j$ precedes job $k$ in $\mathcal{J}_s$ only if $\Delta_j > \Delta_k$, i.e., job $j$ is more likely to violate its due date. As a first randomization step, some jobs in $\mathcal{J}_s$ can be swapped, with probability inversely proportional to their priority.

For every job $j$ in the sorted list $\mathcal{J}_s$, the set of configurations such that the job can be fully processed within its due date is defined, at line 8, as:

$$D_j^* = \{(v,g) \in \mathcal{V} \times \mathcal{G}_v : \ T_c + t_{jvg} < d_j\}.$$

The set $D_j^*$ is used to identify a high-quality configuration for job $j$, according to the following rules. Since one of the main contributions to the schedule total cost is given by the penalties for due date violations, the algorithm always tries to select configurations that guarantee to complete the jobs within the due dates or to minimize the total weighted tardiness. Therefore, the configuration to be selected is:

$$\left(v^*, g^*\right) = \begin{cases} \operatorname{argmin}_{D_j^*} \{t_{jvg} c_v\} & \text{if } D_j^* \neq \emptyset \\ \operatorname{argmin}_{v \in \mathcal{V}, g \in \mathcal{G}_v} \{t_{jvg}\} & \text{otherwise.} \end{cases}$$

The algorithm explores the space of possible configurations by introducing some randomness in the selection process. Instead of choosing the configuration according to the rule defined above, it selects as a candidate configuration for job $j$ one of the $(v,g)$ with lower cost, with probability inversely proportional to the cost itself.

Let $(v^*, g^*)$ be the selected configuration for job $j$ and $\mathcal{N}_O$ the set of open nodes; the assignment proceeds as follows:

1) First, the algorithm tries to assign job $j$ to an open node, following a best-fit approach. Since one of the main goals of our method is to minimize the amount of idle resources, open nodes whose VM type is equal to the one required by the current job are sorted according to the following rule. For each node $\nu \in \mathcal{N}_O$ equipped with a VM of type $v^*$ and such that it hosts enough free resources to assign $g^*$ GPUs to the job $j$, we compute the number $\widehat{g}_\nu$ of GPUs remaining idle after having deployed job $j$ on $\nu$ with the required configuration $(v^*, g^*)$. Nodes are sorted in decreasing order of $\widehat{g}_\nu$, generating a new set $\mathcal{N}_j^* \subseteq \mathcal{N}_O$. Job $j$ should be assigned to a node in $\mathcal{N}_j^*$ with the aim of saturating, as much as possible, the node resources, so that the overall number of idle GPUs is minimized. To introduce randomness, the algorithm assigns $j$ to one of the open nodes $\nu \in \mathcal{N}_j^*$ with probability inversely proportional to the corresponding $\widehat{g}_\nu$.
2) If the assignment to an already open node is not feasible, but $|\mathcal{N}_O| < N$, a new node $\nu'$ is opened, with VM type $v^*$ and the maximum number $G_{v^*}$ of available GPUs. Job $j$ is then assigned to $\nu'$ with configuration $(v^*, g^*)$ and the number of available GPUs on $\nu'$ is updated accordingly.
3) If $(v^*, g^*)$ does not fit in any open node and all nodes are already open ($\mathcal{N}_O = \mathcal{N}$), job $j$ is assigned to a node $\nu \in \mathcal{N}_O$ with a suboptimal configuration. This procedure follows a best-fit approach and assigns the given job to the best among the suboptimal configurations available on open nodes. As in the previous cases, the best suboptimal configuration is the cheapest available configuration that allows the node to execute the job before the due date or, if such a configuration does not exist, the one that minimizes the tardiness.
4) Finally, if also the assignment at the previous point is not feasible because all the resources are saturated, the job remains in the queue until the next scheduling point.

### *Path Relinking Procedure*

Path relinking is typically used as an intensification strategy to enhance the expected quality of results returned by other heuristic methods. This is achieved by exploring trajectories (sequences of alterations in the structure of a solution) connecting good-quality solutions [26], [27]. We adapted this general approach to our context (see Algorithm 2), implementing a procedure that receives as input the set of *elite solutions* $\mathcal{S}^*$ returned by the randomized construction procedure. It extracts from $\mathcal{S}^*$ the candidate solution $S_s$ with best $f_P$ value, generates and explores paths connecting $S_s$ to the other elite solutions in $\mathcal{S}^*$, to find better solutions by incorporating attributes that characterize good-quality solutions.

The procedure is based on the concept of *move*, which denotes any atomic change that can be performed to move from the *source* solution $S_s$ in the direction of a *target* solution $S_t \in \mathcal{S}^*,\ S_t \neq S_s$. The new candidate solution obtained by applying a move $m$ to the current solution $S_s$ is denoted by $S_s \circ m$. In our algorithm, a sequence of moves is performed, either until $S_t$ is reached or until a maximum number of moves is performed. In our context, a move from the source $S_s$ to the target $S_t$ is defined as any pair $(j, n)$ such that job $j$ is assigned to node $n$ in $S_t$, while it is assigned to a different

**Algorithm 2** Path relinking procedure

```
1: function PATH_RELINKING($\mathcal{S}^*$, MaxIt$_{PR}$)
2:     $S_s \leftarrow$ solution in $\mathcal{S}^*$ with best $f_P(S_s)$        ▷ $S_s$: source solution
3:     for all $S_t \in \mathcal{S}^*, S_t \neq S_s$ do        ▷ $S_t$: target solution
4:         iter = 0
5:         while $S_s \neq S_t$ and iter < MaxIt$_{PR}$ do
6:             $\mathcal{M} \leftarrow$ GET_MOVES($S_s, S_t$)
7:             $\mathcal{M}_E \leftarrow \emptyset$        ▷ $\mathcal{M}_E$: set of explored moves
8:             $(m^*, c^*) \leftarrow$ (empty move, $f_P(S_s)$)
9:             for all $m \in \mathcal{M}$ do
10:                if $m \notin \mathcal{M}_E$ then
11:                    $(m^*, c^*) \leftarrow$ EXPLORE_STEP($S_s, S_t, m$)
12:                    $\mathcal{M}_E \leftarrow \mathcal{M}_E \cup \{m\}$
13:                end if
14:            end for
15:            if $m^*$ is not empty then
16:                $S_s \leftarrow (S_s \circ m^*)$
17:            end if
18:            iter $\leftarrow$ iter +1
19:        end while
20:    end for
21:    return $(S_s, f_P(S_s))$
22: end function
```

node $n'$ in $S_s$[2]. More specifically, the function GET_MOVES called at line 6 of Algorithm 2 determines which configuration $(v_t, g_t)$ is assigned to job $j$ in the target solution $S_t$. If this configuration is different in terms of VM type or of number of GPUs from the one selected for job $j$ in the source solution $S_s$, the algorithm looks for a node $n$ that, in solution $S_s$, has VM type $v_t$ and at least $g_t$ free GPUs. If such a node exists, $(j, n)$ is added to the set $\mathcal{M}$ of candidate moves. It is worth noting that either $n'$ or $n$ can be empty, if the job is not executed either in the source solution or in the target one.

Given the set $\mathcal{M}$ of feasible moves, the method explores all possible $m \in \mathcal{M}$ looking for a move that yields a solution with a better proxy function value than $f_P(S_s)$, denoted in line 8 by $c^*$. The exploration of a move $m$, performed at line 11 of Algorithm 2, consists in applying $m$ to the source solution $S_s$, generating a new candidate solution denoted by $S' = S_s \circ m$, and in evaluating the function $f_P$, obtaining $c = f_P(S')$. If the value $c$ is better than the value $c^*$ (i.e., if the current move $m$ yields an improvement), the value of the best move $m^*$, initially empty, is updated accordingly.

To deepen the exploration of the space around $S_s$, the function EXPLORE_STEP is implemented as follows. Instead of considering only the single move $m$ given as input, it proceeds recursively by determining the new set $\mathcal{M}'$ of moves leading from $S' = S_s \circ m$ to the target solution $S_t$ and exploring all $m' \in \mathcal{M}'$ to determine a new candidate solution $S'' = S' \circ m'$. If the value $f_P(S'')$ is better than $c^*$, $m$ becomes the new best move $m^*$. This recursive step strongly affects the quality of the final solution returned by the algorithm, since it allows the procedure to explore a wider range of candidate moves and solutions. In particular, it may happen that a move $m$ would be rejected if we were considering a single-step exploration because $f_P(S') = f_P(S_s \circ m)$ did not provide a better outcome than $c^*$. However, the new candidate solution $S'$ may lead in the second step to a new solution $S'' = S' \circ m'$ such that $f_P(S'')$ is better not only than $f_P(S')$ but also than $c^*$, and

2. Notice that, if the VM type selected on $n$ and $n'$ is the same and job $j$ is executed, in $S_s$ and $S_t$, with the same number of GPUs, the two solutions are equivalent, for our purposes, even if the indices of the nodes are different. Thus, we perform the next steps only when $j$ has a different configuration in the two solutions.

the move $m$ is accepted as new $m^*$. Once the exploration of the set $\mathcal{M}$ is completed, if $m^*$ is not empty, meaning that we found a move providing a better value $c^*$, the solution $S_s$ is updated by applying $m^*$ (see line 16) and the algorithm proceeds with the next iteration, having as new source solution $S_s \circ m^*$.

### *3.3.2 Algorithm Complexity*

Let $J$ be the cardinality of the list of jobs $\mathcal{J}$, $N$ the cardinality of the set of nodes $\mathcal{N}$, and $C = \sum_{v \in \mathcal{V}} G_v$ the cardinality of the set $\mathcal{V} \times \mathcal{G}_v$, i.e., the total number of available configurations. Finally, let $\sigma$ denote the number of candidate good-quality solutions saved in the set $\mathcal{S}^*$. The overall complexity of our method can be written as:

$$\mathcal{O}\left(\text{MaxIt}_{\text{RG}}\left(J \log J + JN \log N\right) + \sigma\ \text{MaxIt}_{\text{PR}}\ M\ J\ N\right).$$

The complete derivation is omitted here for space limits but it is reported in Appendix B.

## 4 Experimental Analysis

We evaluated the heuristic methods proposed in Section 3.3 through an extensive experimental campaign, focusing both on the solution quality and on efficiency. We randomly generated a large set of scenarios as described in Section 4.1. The evaluation methodology, including the description of the alternative approach we adapted from the literature [9], is detailed in Section 4.2 whereas the obtained results are extensively discussed in Section 4.3. As to efficiency, a scalability analysis is reported in Section 4.4. For a fair comparison with alternative methods, we also considered in Section 4.5 scenarios where first-principle methods and the approach proposed in [9] can access a larger number of nodes. Finally, we evaluated Path Relinking in a prototype system deployed on Microsoft Azure; the deviation between the expected and real costs (including VMs and tardiness costs) is discussed in Section 4.6. The algorithm source code and all the results are available as open data at [28].

### 4.1 Randomly Generated Instances

As representatives of long-running DL training jobs, we selected different neural networks training tasks for image and speech recognition, namely Alexnet, Resnet, VGG [22], and DeepSpeech [29], implemented with different DL frameworks (i.e., PyTorch and Tensorflow). A significant heterogeneity characterizes them in terms of resource usage; thus, they can be seen as emblematic samples of the architectures used in practice for image classification and video processing. We considered several instances of the listed DL training workloads, with different epochs and batch sizes and, therefore, different expected execution times, estimated via the Machine Learning models presented in [22] and discussed in Section 2.2. Such models have been trained using data from profiling runs of the target applications, with an average percentage error below 11% (for detailed accuracy results, see [20]).

The considered VM catalog (see Table 2) includes 8 different types. Six of them (NC6, NC12, NC24, NV6, NV12, NV24) are based on Nvidia K80 and M60 and are available on Microsoft Azure. The last ones (NC48* and NV48*) are synthetic VM types obtained from the NC24 and NV24, doubling the number of available GPUs and their hourly costs, in line with the current cloud providers pricing models.

Table 2: Characteristics of the Target Nodes

| **VM type** | **GPU type** | **#GPU** | **Cost [$/h]** |
|---|---|---|---|
| NC6 | K80 | 1 | 0.56 |
| NC12 | K80 | 2 | 1.13 |
| NC24 | K80 | 4 | 2.25 |
| NV6 | M60 | 1 | 0.62 |
| NV12 | M60 | 2 | 1.24 |
| NV24 | M60 | 4 | 2.48 |
| NC48* | K80 | 8 | 4.48 |
| NV48* | M60 | 8 | 4.96 |

To verify the effectiveness and generality of our heuristic method, we generated a set of random problem instances using the parameters described in the following. We varied the number $N$ of available nodes in the cluster from 10 to 100, while the number of jobs in each instance is set to $J = 10N$. Job inter-arrival times were generated as follows:

- In the first instance set, inter-arrivals were drawn, as in other literature proposals (see. e.g., [6]), from an exponential distribution, with mean equal to $75,000s/N$. The mean decreases as the cluster size increases so that the average per-node workload remains almost constant.
- In the other instances, inter-arrival times were generated as described in [9]. Arrivals are sampled from a Poisson distribution, considering three possible rates. Let $\lambda$ be a base rate defined as the reciprocal of the minimum expected completion time given the configurations available in the catalog. The *high* rate is set to $\varepsilon\ k_{max}\ \lambda$, while the *low* rate is $\varepsilon\ k_{max}\ \lambda/4$. We defined $k_{max}$ as the number of nodes in the system multiplied by the maximum number of GPUs that can be assigned to each job. We tuned the parameter $\varepsilon$ to match the peak load of the system to real-life scenarios reported in [30], of nearly 135 job submissions per hour in a system involving few thousands of GPUs. Finally, we obtained the *mixed* rate by alternating *high* and *low* distributions approximately every 10 submissions (similarly to the work in [9]). Since, however, the results achieved under the *mixed* rate are very close to those obtained under exponential inter-arrival times, they are not reported in the following sections but are available at [28].

The distributions of jobs arrivals for a scenario featuring $N = 1000$ and $J = 10000$ are reported in Figure 2. We used the aforementioned job traces to simulate a long-term scenario, involving multiple submissions. The costs are evaluated at the end of the simulation, when all jobs have been completely executed, and include the execution costs depending on the chosen VMs and the tardiness costs of jobs that complete their execution after the due date.

For each value of cluster size and each arrival rate, we generated three problem instances, changing the random seed. The remaining parameters are set as follows. The periodic scheduling time interval $H$ is set to one hour. The due date $d_j$ for each job is sampled from a uniform distribution in the range $[\min_{vg}\{t_{jvg}\},\ 3\min_{vg}\{t_{jvg}\}]$. The tardiness weights $\omega_j$ are extracted with uniform probability from the interval $[0.003, 0.015]$, so that the penalty for a time-unit due date violation is about 10 times larger than the time-unit cost of cloud resources. For what concerns the worst-case tardiness, the parameter $\rho$ that, multiplied by $\omega_j$ in the objective function (see Equation (P1a)), penalizes the postponement of jobs,

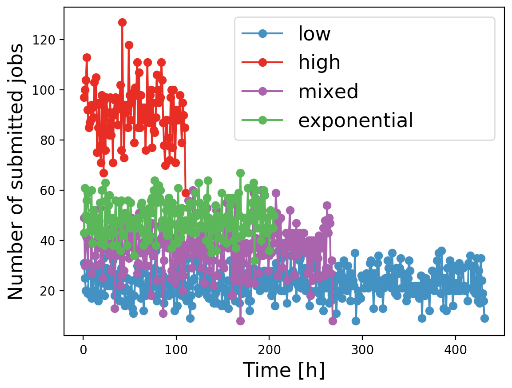


Figure 2: Job submissions under different workloads

is set to 100. Finally, the parameter $\mu$ is set equal to 1 (given the objective function, any positive value forces the use of all the available GPUs).

## 4.2 Evaluation Methodology

As in other literature proposals (see. e.g., [6]), we compared the results of the proposed heuristic methods against those obtained with first-principle methods, namely First in First out (FIFO), Earliest Deadline First (EDF), and Priority Scheduling (PS). Moreover, we performed comparisons against the Hierarchical Method we presented in [11], and with a Dynamic Programming (DP)-based method adapted from [9]. The overall method described in Section 3.3, including the randomized construction process and path relinking, and exploiting the new proxy function presented in Section 3.2, is referred to in the following as Path Relinking algorithm. To highlight the contributions of the different components of our algorithm (ablation study), as mentioned in Section 3.3, we compared it with the results obtained with a Randomized Greedy method, implemented by disabling the path relinking procedure, and with a pure Greedy method, implemented as a deterministic version of the previous one. Both these methods use the original proxy function $f_{\text{OBJ}}$, i.e., Equation (P1a). The results of the Randomized Greedy and pure Greedy methods with the alternative proxy function $\bar{f}_{\text{OBJ}}$ defined in Equation (1), as well as the results of the Path Relinking method exploiting $f_{\text{OBJ}}$ (leading to lower quality solutions) are not reported for space limits.

The comparison is performed by considering the total costs given by the sum of the VM usage costs and the tardiness costs for due date violations, computed after executing all jobs, and the *percentage cost reduction* obtained by Path Relinking with respect to any other method. The latter is defined as:

$$\text{pcr} = \frac{C_{algo} - C_{PR}}{C_{algo}} \cdot 100\%, \tag{3}$$

where $C_{PR}$ denotes the cost obtained by Path Relinking, and $C_{algo}$ is the cost achieved by any other method. Note that, to be conservative, we decided to divide the difference between costs at the numerator by $C_{algo}$, which is expected to be larger than $C_{PR}$ in all the considered scenarios.

Details about how we adapted the DP-based method to our problem, as well as about software and hardware settings, are provided in the next sections.

### 4.2.1 A Dynamic Programming-Based Alternative Method

To assess the effectiveness of our approach, we adapted to our problem a Dynamic Programming (DP)-based method initially proposed in [9] and from now on referred to as *DP algorithm*. [9] presents a resource allocation strategy for DL training jobs that leverages DP to determine, for each job, the number of GPUs to be allocated to it and its optimal batch size. Jobs are then scheduled by applying a FIFO mechanism, such that one single job can be executed on each node. The optimizer is called periodically, with time period $\Delta t$, and jobs execution can be stopped and resumed from a checkpoint to account for changes in the allocated resources.

We extended the DP algorithm by introducing the selection of the VM type $v \in \mathcal{V}$ for each available node $n \in \mathcal{N}$, which was considered as a physical, non-virtualized machine in [9].

The DP algorithm works as follows. For all jobs $j \in \mathcal{J}$:

1) the optimal configuration is selected according to the value of a proxy function $\mathcal{F}_1$ (defined in the following),
2) the cost of executing the job with the selected configuration and the cost of postponing the job to the following scheduling step are computed according to a possibly different proxy function $\mathcal{F}_2$,
3) for all numbers $\hat{n}$ of available nodes in $\mathcal{N}$, the total cost due to the execution or postponement of the job is computed, assuming that $\hat{n} - 1$ nodes are already used to run all the previous jobs. The optimal choice for the job is selected accordingly.

The modified DP algorithm considers the binary variables $y_{nv}$, which, as mentioned in Section 3.1, assume value 1 if the VM type $v \in \mathcal{V}$ is selected on node $n \in \mathcal{N}$, and $z_{jn}$, which is 1 if job $j \in \mathcal{J}$ is deployed on node $n$. Moreover, we introduce, for each job $j \in \mathcal{J}$, a new binary variable $r_j$, which is equal to 1 if $j$ is not executed in the current scheduling step.

To guarantee a fair comparison with our method, we implemented several versions of the DP algorithm, which differ in terms of the proxy functions $\mathcal{F}_1$ and $\mathcal{F}_2$.

The first version, denoted as *DP(WCT)*, is characterized by $\mathcal{F}_1 = \mathcal{F}_2 = \mathcal{F}_{WCT}$, where:

$$\mathcal{F}_{WCT} = \sum_{j \in \mathcal{J}, v \in \mathcal{V}, n \in \mathcal{N}} z_{jn} y_{nv} \omega_j \tau_j + \sum_{n \in \mathcal{N}, v \in \mathcal{V}} y_{nv} c_v \Delta t + \sum_{j \in \mathcal{J}} r_j \hat{\tau}_j. \tag{4}$$

It mimics our proxy function defined in Section 3.1, since the first term represents the penalty for due date violations, the second one measures the operational costs related to VMs usage, and the third one is used to penalize the postponement of jobs via the worst-case tardiness $\hat{\tau}_j$.

We developed alternative proxy functions to enhance the performance of DP(WCT). In particular, focusing on the selection of the best setup, we defined $\mathcal{F}_1$ to select as optimal configuration the one that guarantees the lowest execution time. This is particularly effective in high-load scenarios, which are the most challenging ones as pointed out in the next sections. Such choice was coupled with two alternatives for $\mathcal{F}_2$. In the first version, denoted as *DP(FastWCT)*, we kept $\mathcal{F}_2 = \mathcal{F}_{WCT}$. In the second version, denoted instead as *DP(FastB)*, we defined $\mathcal{F}_B$ by replacing the worst-case tardiness $\hat{\tau}_j$ from Equation (4) with a positive constant $B$. Indeed, if the due date of a job $j$ is very large, the corresponding $\hat{\tau}_j$ becomes 0, which means that the job may be postponed with no impact on the proxy function value. This, however, affects the performance in the long term, since postponed jobs risk to violate their due dates if resources are not enough to execute them in the near future.

Finally, we decided to couple $\mathcal{F}_2 = \mathcal{F}_{WCT}$ with a modified function $\mathcal{F}_1$ given by:

$$\tilde{\mathcal{F}} = \sum_{j\in\mathcal{J},v\in\mathcal{V},n\in\mathcal{N}} z_{jn}y_{nv}\omega_j\tilde{\tau}_j + \sum_{n\in\mathcal{N},v\in\mathcal{V}} y_{nv}c_v\Delta t + \sum_{j\in\mathcal{J}} r_jB. \quad (5)$$

The first term of Equation (5) is obtained by substituting the tardiness $\tau_j$ with an *adjusted tardiness* $\tilde{\tau}_j = \max\{0, T_c + t_{jvg} - d_j\}$, where $T_c$ is the current time and $t_{jvg}$ is the execution time of job $j$ with the configuration it is assigned to. This updated term measures the delay of job $j$ with respect to its due date if it is fully executed with the current configuration (i.e., assuming that no migration will occur in the following scheduling steps), thus penalizes slow configurations even if no tardiness occurs at the end of the current scheduling step. The method obtained exploiting $\mathcal{F}_1 = \tilde{\mathcal{F}}$ and $\mathcal{F}_2 = \mathcal{F}_{WCT}$ is denoted by *DP(AdjWCT)*.

#### 4.2.2 *Software and Hardware Settings*

As mentioned above, the Hierarchical Method partitions the set of jobs into $K$ subsets and solves in parallel the MILP formulation presented in Section 3 for small subsets of nodes and jobs, using a state-of-the-art solver. In the experiments, Gurobi Optimizer 9.0 is used, with the mixed-integer programming gap (i.e., the difference between the current upper and lower bounds of the MILP solver) set to 5%. We considered a number of local controllers $K = N/5$, so that each local controller has to manage a system of fixed size, involving 5 nodes and 50 jobs, and solving each MILP formulation required between 1.5 and 33 seconds on average in the different scenarios.

The first-principle methods and the heuristic algorithm described in Section 3.3, as well as the DP-based methods presented in Section 4.2.1, are implemented in C++. This guarantees, as discussed in Section 4.4, good scalability properties and fast execution times. As for the Randomized Greedy and Path Relinking algorithms, we set $\text{MaxIt}_{\text{RG}} = 1000$, $\text{MaxIt}_{\text{PR}} = N$, and the number $\sigma$ of elite solutions built in the randomized construction procedure to 10. For the DP-based methods, we tested different values of the time elapsed between two scheduling steps, $\Delta t$. For the sake of space, we discuss in the following section only the best results, obtained with $\Delta t = 15\ min$.

We performed the experiments with 10 different random seeds for each job trace, for a total number of nearly 700 tests. The server time required to complete the experimental campaign (on an Ubuntu 18.04 VM based on a dual Intel Xeon Silver 4114 CPU at 2.20GHz with overall 40 cores and 64GB of memory) is of about one week.

### 4.3 Experimental Results

To compare the results obtained with the different methods, we considered, as previously mentioned, the total costs given by the sum of the VM usage costs and the tardiness costs for due date violations, computed after executing all jobs. This approach guarantees fair comparisons across the different methods, even if they pursue the optimization of different proxy functions. Moreover, we computed the ratio between the average total cost obtained with all the proposed methods for each considered scenario and the average total cost of Earliest Deadline First (EDF), which is, among the first-principle methods, the one yielding the best performance. The results under exponential inter-arrival times, *high* and *low* rates are shown in Figures 3a, 3b and 3c, respectively, where we can observe that the Path Relinking method leads to solutions with the lowest costs in all analyzed scenarios.

For what concerns the percentage cost reduction (pcr) defined in Equation 4.2, Figure 4 reports the minimum (orange bar), maximum (green bar) and average (blue bar) value of the pcr obtained across all instances, under different load scenarios. We observe that the Path Relinking method achieves a significant percentage cost reduction, between 23% and 97% on average, with respect to EDF, between 7 and 20% compared with the Hierarchical Method, and between 43 and 95% against the best among the versions presented in Section 4.2.1 (DP(AdjWCT) in the *low* rate scenario, DP(FastB) in all the others). The fact that, in the *low* rate scenario, all heuristic methods variants obtain a less significant percentage cost reduction when compared to EDF (of about 23% on average for Path Relinking) is motivated by the fact that, in this context, the system load is reduced; consequently, it is easier to meet the due dates even with simpler strategies.

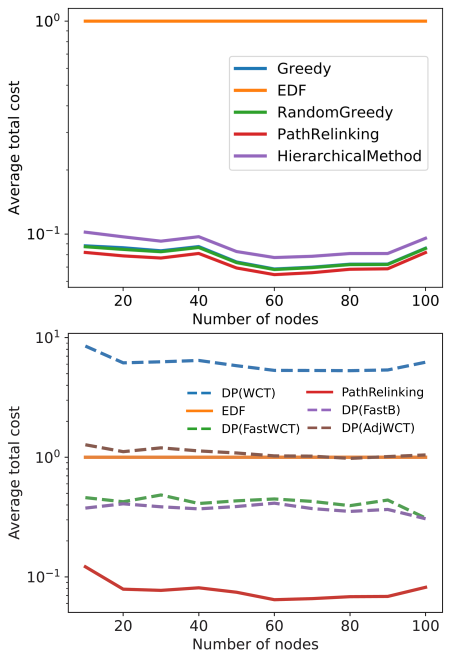
We performed the Analysis of Covariance (ANCOVA) [21], [31]–[33] to assess the statistical significance of the observed differences among the proposed algorithms. The extensive discussion of the tests is not reported here for space limits, but it is available as Appendix C in the additional material.

### 4.4 Scalability Analysis

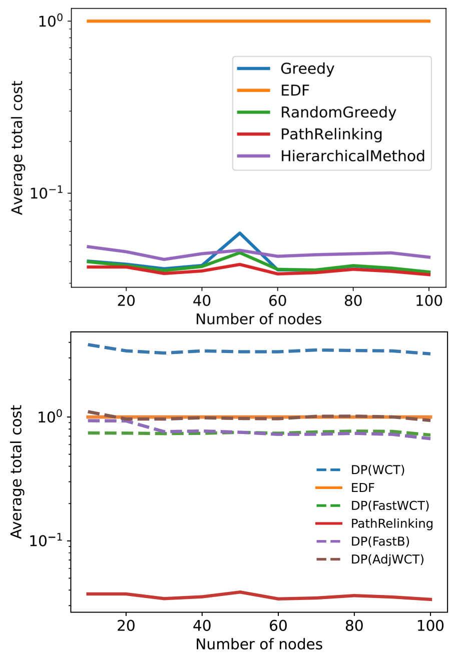
To better evaluate the performance of the proposed heuristic method and its simplified variants, we analyzed the time required to solve a single instance of the problem formalized in the previous sections. By inspection, we observed that the maximum number of concurrent jobs in a system with 100 available nodes is around 450 for the *high* rate scenario, and around 250, 200 and 150 for the *exponential*, *mixed* and *low* rates, respectively. The results in all the considered scenarios are shown in Figure 5. Notice that Path Relinking, Randomized Greedy and pure Greedy, albeit addressing the full problem, exhibited a lower execution time than what required by each local controller of the Hierarchical Method in all the scenarios and independently of the problem size. The speed-up obtained with the Path Relinking algorithm is of one order of magnitude on average when considering exponential inter-arrival times and when considering *high* or *mixed* rates. It is slightly reduced in the case of *low* rate, since the optimization problem instances are usually easier to solve. An even more significant speed-up, between 3 and 4 orders of magnitude on average, is obtained by the pure Greedy algorithm, which, as shown in Section 4.3, yields solutions whose quality is comparable to those produced by the Hierarchical Method.

### 4.5 Analysis with a Larger Number of Nodes

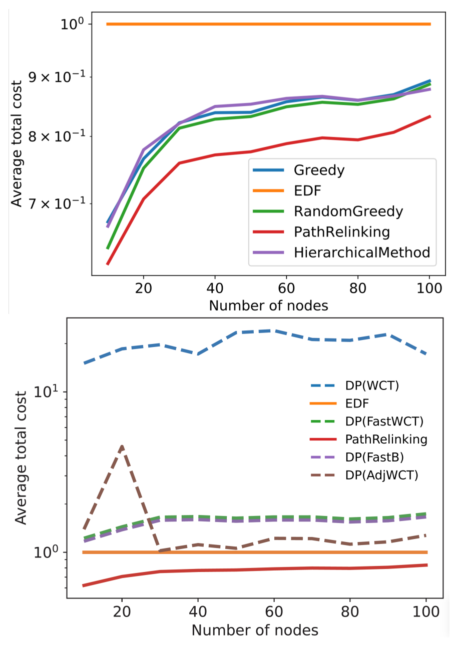
First-principle methods as EDF, as well as the dynamic programming-based algorithms of Section 4.2.1, work under the assumption that only one job can be deployed on each node. Moreover, EDF does not allow jobs rescheduling: the resources that they receive remain fixed until the complete execution. These aspects may have a strong impact on the



(a) Exponential inter-arrival times

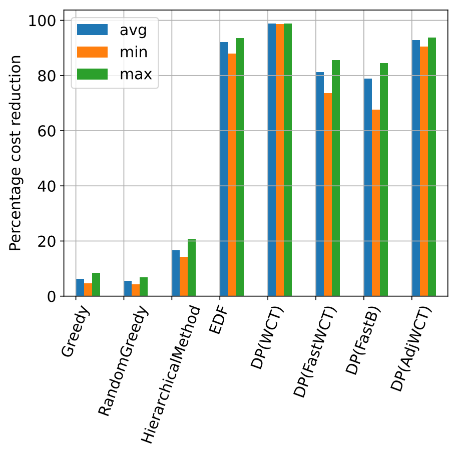


(b) High rate

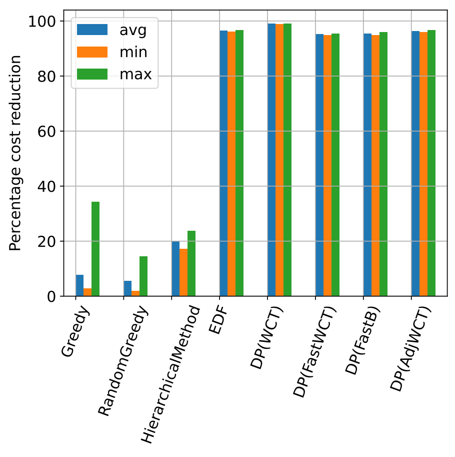


(c) Low rate

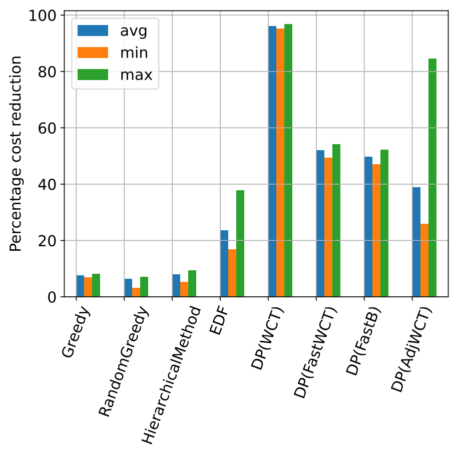
Figure 3: Average total costs obtained with all methods



(a) Exponential inter-arrival times

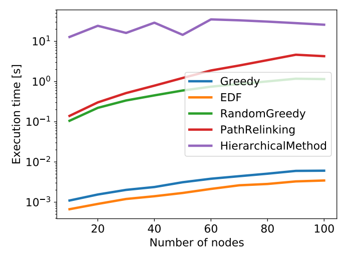


(b) High rate

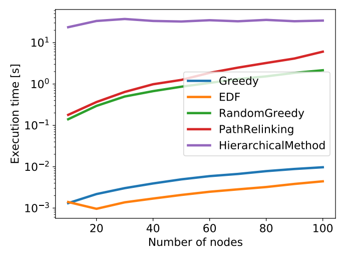


(c) Low rate

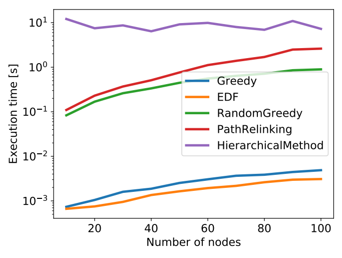
Figure 4: Percentage cost reduction obtained by Path Relinking with respect to the other methods



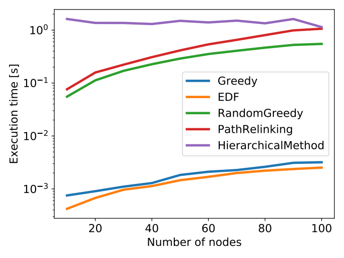
(a) Exponential inter-arrival times



(b) High rate



(c) Mixed rate



(d) Low rate

Figure 5: Average execution times with the different methods

system costs, since under heavy load resources get saturated quickly, often leading to due date violations. To mitigate the effect of such constraints, which limit the overall number of GPUs that may be used, we compared the result achieved by our methods in each run exploiting $N$ available nodes with the ones obtained by EDF, DP(WCT), DP(FastWCT), DP(FastB) and DP(AdjWCT) in a scenario involving the same job trace, but $2N$, $4N$ or $8N$ nodes. Specifically, systems with $8N$ nodes were considered because 8 is the maximum number of GPUs available in a VM and, therefore, at most 8 jobs can be co-located in a node by our algorithms.

The detailed plots of average total costs and percentage cost reductions obtained with the different methods are reported in Appendix D. It is worth noting that the results in the $4N$ and $8N$ scenarios are almost identical: the number of available nodes in the $4N$ setting is large enough to execute all the concurrent jobs, therefore the proposed solution found is the best solution that can be achieved by the different algorithms. Moreover, in the $4N$ and $8N$ scenarios the DP(AdjWCT) method always yields lower costs with respect to the other variants presented in Section 4.2.1. Indeed, the adjusted tardiness $\tilde{\tau}_j$ considered in the objective function leads to a more effective resource management: when there are enough available resources to run all jobs, there is no need to necessarily select the fastest configuration (which is usually the most expensive), thus DP(FastB) and DP(FastWCT) are less suited to such scenarios. In the $2N$ case, this happens in the *mixed* and *low* rate settings, where the system load is lower, while it does not happen in the *high* rate setting, because the system is still subject to a higher pressure.

Despite the larger amount of resources considered by EDF and the DP-based methods, Path Relinking yields better results in all the considered scenarios, even if the average percentage gain is reduced, in the worst case, from 96% to 10% with respect to EDF and from 39% to 8% with respect to DP(AdjWCT).

It is worth noting that exploiting a higher number of nodes could be financially burdensome, especially when relying on reserved instances, which include additional yearly costs to have a reduced resources hourly cost (as an example, the AWS entry level T4 instances have an upfront payment of about 1000 USD per year per node at the time of writing [34]). Therefore, methods that yield lower or equivalent costs requiring less resources are to be preferred.

## 4.6 Validation in a Cloud Cluster

To validate the results obtained by the Path Relinking method, we deployed on Microsoft Azure a prototype system including six different applications. Their characteristics in terms of network types, epochs and batch sizes are summarized in Table 3. We set the inter-arrival time to $300s$, while we generated the due dates $d_j$ and tardiness weights $\omega_j$ for all jobs as described in Section 4.1. The considered scenario is accelerated, in terms of submission frequency and average execution times of applications, to limit cloud operations costs. This does not affect the solution effectiveness since the workload assigned to the available nodes is comparable to practical situations. The system is composed of two nodes, while the considered cloud provider catalog includes the first six types of VMs, with two types of GPUs, listed in Table 2.

The results provided by the Path Relinking algorithm are reported in Figure 6. The schedule obtained at each step was implemented and run on a real system to compare the expected performance with the actual one. A discrepancy in execution and completion times of jobs was expected because the time required to deploy and boot the VMs on the system and the time required to migrate applications from one VM instance to another are not negligible due to the accelerated time setting of the experiment. Moreover, the execution times used to build the schedule and to compute the expected results were are estimated, as reported in Section 4.1, through Machine Learning models and are, therefore, prone to errors.

Table 3: Applications submitted to the system

| Job | Application | Files | Epochs | Batchsize |
|---|---|---|---|---|
| JJ1 | DeepSpeech (TensorFlow) | 64 | 158 | 4 |
| JJ2 | VGG19 (PyTorch) | 130,000 | 1 | 32 |
| JJ3 | VGG19 (TensorFlow) | 130,000 | 1 | 32 |
| JJ4 | AlexNet (PyTorch) | 130,000 | 12 | 256 |
| JJ5 | ResNet50 (PyTorch) | 130,000 | 3 | 64 |
| JJ6 | ResNet50 (TensorFlow) | 130,000 | 3 | 64 |

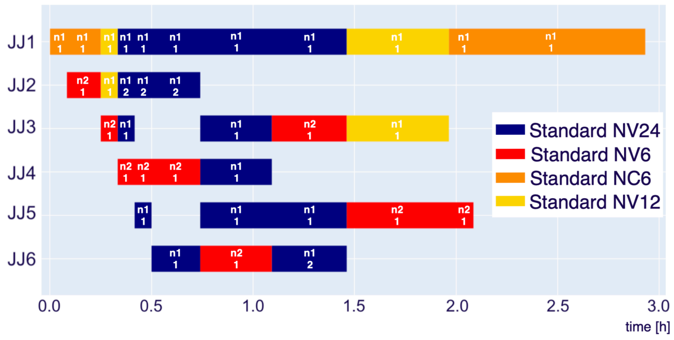


Figure 6: Real System - For each job, the index of the node and the number of assigned GPUs are reported one on top of the other.

Table 4: Predicted and actual costs on the system

| Slot | Predicted costs [\$] | Actual costs [\$] |
|---|---|---|
| 1 | 0.09 | 0.09 |
| 2 | 0.39 | 0.39 |
| 3 | 0.32 | 0.32 |
| 4 | 0.53 | 0.53 |
| 5 | 0.53 | 0.53 |
| 6 | 1.54 | 1.88 |
| 7 | 2.72 | 2.47 |
| 8 | 2.37 | 2.39 |
| 9 | 3.33 | 3.68 |
| 10 | 1.28 | 1.37 |
| 11 | 2.93 | 3.06 |
| **Sum** | 16.03 | 16.72 |
| **Difference** | | 4.31% |

A comparison between the predicted and actual costs in all scheduling slots is reported in Table 4. Notice a difference between the predicted and the actual values due to a discrepancy in all slot durations starting from the sixth one. The difference, lower than 5%, can be motivated by the mentioned overhead due to the rebooting of the VMs and to the fact that training jobs migrated from one VM to another must be re-executed from the last checkpoint. Furthermore, this deviation is computed considering a small system that run less than three hours, while the training time for real applications is, usually, considerably longer. Errors attributable to rebooting and migration times appear less significant in larger settings, promoting the applicability of the proposed approach.

## 5 Related Work

Considering the relatively slow increase of per-core computation power witnessed in the last decade, the natural way to get an actual computation speed-up nowadays is to resort to a higher degree of parallelism, like the one achievable using one or more GPUs. Nevertheless, while GPU farms deliver unprecedented computing power, such potential is still difficult to harness [35], and new challenges arise in GPU as a Service environments. Among others, job scheduling calls for both a robust theoretical framework and viable, practical solutions [36]. To the best of our knowledge, our work represents one of the first attempts to tackle the joint problem of online DL job scheduling and resource allocation on multiple virtualized GPUs. As mentioned in Section 1, most of the available literature proposals focus on either the scheduling or the resource selection aspect, leaving the decisions on the number and type of GPUs to the users (e.g., [6], [7]) or delegating the job scheduling to simple mechanisms as FIFO or EDF (e.g., [8], [9]), respectively. Therefore, we will briefly review part of the existing literature in the two scenarios.

### *Jobs Scheduling*

A greedy algorithm for GPU workloads scheduling is proposed in [6]. It exploits two graphs to represent jobs communication requirements and the GPU topology, and allows jobs co-location on the basis of performance interference, preferentially placing as many jobs tasks as possible on the same node to reduce communication overheads. Gandiva [7] is mainly focused on workloads characterized by multiple submissions, where users exploit early feedback to dynamically prioritize or kill a subset of jobs. The proposed scheduler operates in reactive mode, exploiting a job placement policy with over-subscription to deal with job arrivals, departures and failures, and/or in introspective mode, continuously monitoring and adapting jobs placement to improve GPUs and nodes usage and reduce jobs completion time. The same goal is pursued by Harmony [17], a deep reinforcement learning-based scheduler that evaluates the impact of co-location to reduce interference. The minimization of renting costs in public cloud, which is the goal of our approach, is considered also in [37]. However, in this work jobs are subject to hard deadlines and Virtual Machines are rented from the public cloud to scale the available resources and avoid violations. A data-driven Dynamic Voltage Frequency Scaling method is exploited in [8] to guide a deadline-aware scheduling algorithm based on an EDF approach, aiming to maximize the energy efficiency of a cluster. Inter-user fairness is the main goal pursued by Gandiva$_{fair}$ [14]. It exploits a central, gang-aware scheduler for large jobs that span multiple servers, and a local, per-server, gang-aware scheduler for small jobs. Job migration is considered as a key to achieve load balancing, since it allows to pack multiple applications that require a single GPU on the same server when resources become available. Fairness is crucial also for Themis [18], where a round-by-round partial allocation auction is exploited to allow applications to specify their placement preferences, providing Pareto efficiency and maximizing sharing incentive.

Finally, $\rho$ competitive algorithms for online non-clairvoyant job scheduling and resource allocation problems in virtualized clusters are studied in [38] and [39]. However, the problems considered, while having some features in common with the one addressed in this paper, are considerably simpler and thus more suitable to be analyzed with the tool of competitive analysis [40]. In this framework, online algorithms are analyzed and ranked based on their worst-case behavior compared to the solution of an optimal offline algorithm. Furthermore, the evaluation must consider an infinite set of instances. For this reason, the competitive analysis applies only to small-size and easily characterizable problems. An alternative approach entails using simulation [41] to estimate the difference between the optimal offline value of the objective function and the expected value of the online algorithm. The latter approach is not feasible either since the optimal solution for a complex optimization problem like the one presented in this paper cannot be found in a reasonable computing time with the currently available technologies, even at each rescheduling point.

### *Resource Selection*

For what concerns resource allocation of DL jobs, DL$^2$ [30] combines an offline supervised learning and an online reinforcement learning-based approach for resource selection, setting the number of workers/parameter servers (PS) to adopt for DL training jobs. The MXNet framework is improved to support the dynamic "hot" scheduling, i.e., to adapt the resource assignment without stopping the jobs' execution. Optimus [15] proposes a mathematical formulation coupled with a heuristic algorithm based on marginal gains to dynamically allocate the number of workers/PS that minimizes the jobs' completion time, while the amount of resources required by each worker/PS is specified by the user. In order to reduce communication overheads, jobs are executed on the minimum number of servers that allow to place an equal amount of workers/PS. A resource allocation strategy for DL training jobs is proposed in [9]. The authors develop an optimization formulation where the optimal job batch size is set according to their scaling efficiency. Moreover, they propose a dynamic programming-based heuristic algorithm to determine an effective resource allocation, while jobs are scheduled relying on a FIFO mechanism. Finally, an interference-aware and prediction-based resource manager is proposed in [19], where GPU utilization is identified as a proxy metric that allows to determine good placement decisions.

## 6 Conclusions

This paper proposes a heuristic method, developed by integrating randomized greedy and path relinking algorithms, to tackle the online joint capacity allocation and scheduling problem for DL training jobs in GPU as a Service systems. An extensive experimental campaign proves the effectiveness of the proposed method both in terms of scalability and solution quality. Significant cost savings, between 23 and 97% on average, were obtained compared to first-principle methods in all the considered scenarios. Moreover, an average percentage cost reduction between 7 and 20% is attained with respect to the Hierarchical Method in [11], and between 43 and 95% against a dynamic programming-based method adapted from [9]. The scalability analysis shows that the solution for systems with up to 100 nodes and 450 concurrent jobs can be computed in less than 7 seconds, proving the effectiveness of our approach for practical scenarios. Validating our results in a cloud prototype environment showed a deviation between real and predicted costs below 5%.

In our research agenda we plan to consider the joint capacity allocation and scheduling problem in data center environments with disaggregated resources, as well as in distributed cloud-edge architectures. Future work will also consider scenarios where GPU sharing among multiple jobs and distributed training across multiple servers are allowed. Finally, for a simplified version of the problem an analysis to characterize the optimality gap of the proposed solution could be also performed.

## Acknowledgments

Federica Filippini and Danilo Ardagna's work has been funded by the European Commission under the H2020 grant N. 101016577 AI-SPRINT: AI in Secure Privacy pReserving computINg conTinuum.

## References

[1] L. Deng, G. Li, *et al.*, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[2] GlobalMarketInsights, "Gpu as a service market size by product," *[Online]. Available: https://www.gminsights.com/industry-analysis/gpu-as-a-service-market (visited on 04/10/2021)*,

[3] R. E. Shawi, A. Wahab, *et al.*, "Dlbench: A comprehensive experimental evaluation of deep learning frameworks," *Cluster Computing*, vol. 24, no. 3, pp. 2017–2038, 2021.

[4] V. Anand, "Nvidia dgx a100," *[Online]. Available: https://www.hardwarezone.com.sg/tech-news-nvidia-dgx-a100-supercomputer-super-performance-fight-covid-19 (visited on 04/10/2021)*,

[5] NVIDIA, "Nvidia virtual gpu technology," *[Online]. Available: https://www.nvidia.com/en-us/design-visualization/technologies/virtual-gpu/ (visited on 04/10/2021)*,

[6] M. Amaral, J. Polo, *et al.*, "Topology-aware gpu scheduling for learning workloads in cloud environments," in *HPCNSA Proc.*, ACM, 2017.

[7] W. Xiao, R. Bhardwaj, *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *USENIX OSDI*, 2018.

[8] S. Ilager, R. Muralidhar, *et al.*, "A data-driven frequency scaling approach for deadline-aware energy efficient scheduling on graphics processing units (gpus)," in *IEEE/ACM CCGRID*, 2020, pp. 579–588.

[9] V. Saxena, K. R. Jayaram, *et al.*, "Effective elastic scaling of deep learning workloads," in *MASCOTS proceedings*, 2020, pp. 1–8.

[10] M. Gendreau and J.-Y. Potvin, Eds., *Handbook of Metaheuristics*, ser. International Series in Operations Research & Management Science. Springer International Publishing, 2019.

[11] F. Filippini, M. Lattuada, *et al.*, "Hierarchical scheduling in on-demand gpu-as-a-service systems," in *SYNASC2020*, 2020, pp. 125–132.

[12] *Amazon web service pricing list*, 2021. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls.

[13] *Azure cloud services pricing list*, 2021. [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/cloud-services/.

[14] S. Chaudhary, R. Ramjee, *et al.*, "Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning," in *EUROSYS*, 2020.

[15] Y. Peng, Y. Bao, *et al.*, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *EUROSYS*, 2018.

[16] H. Larochelle, D. Erhan, and Y. Bengio, "Zero-data learning of new tasks," vol. 2, Jan. 2008, pp. 646–651.

[17] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," in *IEEE INFOCOM 2019*, Apr. 2019, pp. 505–513.

[18] K. Mahajan, A. Balasubramanian, *et al.*, "Themis: Fair and efficient GPU cluster scheduling," in *USENIX (NSDI 20)*, 2020, pp. 289–304.

[19] G. Yeung, D. Borowiec, *et al.*, "Horus: Interference-aware and prediction-based scheduling in deep learning systems," *IEEE TPDS*, vol. 33, no. 1, pp. 88–100, 2022.

[20] M. Lattuada, E. Gianniti, *et al.*, "Performance prediction of deep learning applications training in GPU as a service systems," *Cluster Computing*, vol. 25, no. 2, pp. 1279–1302, 2022.

[21] G. James, D. Witten, *et al.*, *An Introduction to Statistical Learning*, ser. Springer Texts in Statistics. Springer-Verlag New York, 2013.

[22] E. Gianniti., L. Zhang., and D. Ardagna., "Performance prediction of gpu-based deep learning applications," in *CLOSER*, vol. 1, 2019, pp. 279–286.

[23] M. Jeon, S. Venkataraman, *et al.*, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads," in *USENIX ATC 19*, 2019, pp. 947–960.

[24] Q. Hu, P. Sun, *et al.*, "Characterization and prediction of deep learning workloads in large-scale GPU datacenters," *CoRR*, vol. abs/2109.01313, 2021.

[25] M. R. Azghadi, C. Lammie, *et al.*, "Hardware implementation of deep network accelerators towards healthcare and biomedical applications," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 14, no. 6, pp. 1138–1159, 2020. DOI: 10.1109/TBCAS.2020.3036081.

[26] M. G. Resende and C. C. Ribeiro, "Grasp with path-relinking: Recent advances and applications," in *Metaheuristics: Progress as Real Problem Solvers*. Springer US, 2005, pp. 29–63.

[27] M. Resende and C. Ribeiro, "Greedy randomized adaptive search procedures: Advances and applications," *Handbook of Metaheuristics*, Jan. 2010.

[28] F. Filippini, M. Lattuada, *et al.*, *Ai-sprint gpu scheduler*, version V1, Zenodo, Dec. 2021. DOI: 10.5281/zenodo.5760962. [Online]. Available: https://doi.org/10.5281/zenodo.5760962.

[29] A. Hannun, C. Case, *et al.*, *Deep speech: Scaling up end-to-end speech recognition*, 2014. arXiv: 1412.5567.

[30] Y. Peng, Y. Bao, *et al.*, "Dl2: A deep learning-driven scheduler for deep learning clusters," *IEEE TPDS*, vol. 32, no. 08, pp. 1947–1960, 2021.

[31] B. Huitema, *The Analysis of Covariance and Alternatives: Statistical Methods for Experiments, Quasi-*

*Experiments, and Single-Case Studies*, ser. Wiley Series in Probability and Statistics. Wiley, 2011.

[32] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[33] G. W. Snedecor and W. G. Cochran, *Statistical Methods - 8th Edition*. Iowa State University Press, 1989.

[34] *Amazon ec2 reserved instances pricing*, 2021. [Online]. Available: https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/.

[35] M. Steinberger, "On dynamic scheduling for the gpu and its applications in computer graphics and beyond," *IEEE CGA*, vol. 38, no. 3, pp. 119–130, 2018.

[36] H. Tan, Y. Tan, *et al.*, "A virtual multi-channel gpu fair scheduling method for virtual machines," *IEEE TPDS*, vol. 30, no. 2, pp. 257–270, 2019.

[37] J. Zhu, X. Li, *et al.*, "Scheduling stochastic multi-stage jobs to elastic hybrid cloud resources," *IEEE TPDS*, vol. 29, no. 6, pp. 1401–1415, 2018.

[38] S. M. Khorandi and M. Sharifi, "Non-clairvoyant online scheduling of synchronized jobs on virtual clusters," *The Journal of Supercomputing*, vol. 74, no. 6, pp. 2353–2384, 2018.

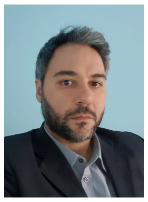
[39] Y. Li, X. Tang, and W. Cai, "Dynamic bin packing for on-demand cloud resource allocation," *IEEE TPDS*, vol. 27, no. 1, pp. 157–170, 2015.

[40] A. R. Karlin, M. S. Manasse, *et al.*, "Competitive snoopy caching," *Algorithmica*, vol. 3, no. 1, pp. 79–119, 1988.

[41] F. Dunke and S. Nickel, "Simulative algorithm analysis in online optimization with lookahead," *Simulation in Produktion und Logistik: Entscheidungsunterstützung von der Planung bis zur Steuerung*, pp. 405–416, 2013.

**Federica Filippini** is a Ph.D. student at Politecnico di Milano where she received in April 2020 the Master degree in Mathematical Engineering. Her research interests include optimization problems applied to scheduling in Cloud and distributed environments.

**Marco Lattuada** received the Master and the Ph.D. degrees in Computer Engineering from Politecnico di Milano in 2006 and 2010 respectively. His research interests include methodologies for performance estimation of big data applications running on cloud cluster and methodologies for performance estimation and automatic generation of code for multiprocessor embedded heterogeneous architectures.



**Michele Ciavotta** received the Ph.D. degree in automation and computer science from Roma Tre, Italy in 2008. He is researcher at the University of Milano - Bicocca since 2017. His research work focuses on the modeling and optimization of highly constrained combinatorial problems mainly arising in the fields of scheduling and resource management of distributed systems.

**Arezoo Jahani** is Assistance Professor at Sahand University of Technology (SUT), Tabriz, Iran. She received the Ph.D. degree in Information Technology in 2019 and master degree in Computer Engineering in 2014, from University of Tabriz. Her research interests include virtual network embedding in data centers, optimization algorithms, and cloud resource management.

**Danilo Ardagna** is Associate Professor at Politecnico di Milano, DEIB. He received a Ph.D. degree in computer engineering in 2004 from Politecnico di Milano. His work focuses on the design, prototype, and evaluation of optimization algorithms for resource management of cloud computing and big data systems.

**Edoardo Amaldi** is a Full Professor of Operations Research at Politecnico di Milano, DEIB. He received a "Diplôme" (M.Sc.) in Mathematical Engineering and a "Doctorat ès Sciences" (Ph.D.) from the Swiss Federal Institute of Technology (EPFL). His main research interests are in Mathematical Optimization with an emphasis on Network Optimization and with applications in, among others, telecommunications, data mining, energy and transportation.