Scheduling Deep Learning Jobs Training in the Cloud: Comparing Multiple Approaches

Michele Precuzzi, Federica Filippini, Danilo Ardagna
Dipartimento di Elettronica Informazione e Bioingeneria
michele.precuzzi@mail.polimi.it, federica.filippini@polimi.it, danilo.ardagna.polimi.it

Abstract

Deep learning (DL) methods have gained popularity in recent years. However, it is well known that training this class of models is often computationally expensive. Graphic Processing Units (GPUs) are frequently used to boost performance and, while the cloud remains the most cost-effective and flexible deployment, the overall cost can be consistently reduced by efficiently sizing and sharing the available resources among the various processes.

This work addresses the online joint capacity planning and job scheduling with deadlines problem for DL training jobs. In particular, we compare our previously proposed Hierarchical approach with a new Dynamic-Programming-based algorithm, adapted from a State-of-the-Art method, and with a First-Principle method (Earliest-Deadline-First) in terms of efficiency and computational costs. Our experimental campaign proves the efficiency of the Hierarchical method, which achieves an average percentage cost reduction between 16% and 96% with respect to the Earliest-Deadline-First approach, and between 36% and 99% compared to the Dynamic-Programming-based algorithms.

1 Introduction

The widespread adoption of Graphic Processing Units (GPUs) to boost the training performance of Neural Networks (NNs)-based applications helped the Deep Learning (DL) paradigm in gaining increasing popularity. DL is exploited in many fields as, e.g., language recognition (Wang et al. 2019) and brain cancer detection (Ranjbarzadeh et al. 2021). It greatly benefits from GPU acceleration thanks to highly efficient linear algebra libraries. However, training DL applications is still computationally demanding, and the adoption of GPU acceleration on a large scale is extremely expensive even relying on pay-per-use cloud pricing models (Cao et al. 2019), resulting as a feasible approach for large organizations only. In light of this, today's research is focusing more and more on developing methods that allow an efficient resources management in order to obtain a significant costs reduction. In this paper, we compare our previous work presented in (Filippini et al. 2020), based on Hierarchical optimization and hereafter denoted as Hierarchical method, with a new Dynamic Programming (DP)-based algorithm, adapted from a State-of-the-Art method initially proposed in (Saxena et al. 2020).

The Hierarchical method considers multiple DL training jobs, continuously submitted for execution on a cluster of cloud nodes. Individual nodes can be configured from a variety of Virtual Machine (VM) types available from the cloud provider's catalog, such that each type features, possibly, several GPUs. Each node can be configured with a single VM type and multiple job can run on it.

Each job is characterized by a deadline, a batch size and a tardiness cost (i.e., a penalty cost proportional to the difference between the job completion time and its deadline, and its priority). The set of jobs to be scheduled is not known in advance: new jobs are submitted with different characteristics, deadlines, and priorities without any repetition scheme, resulting in an online problem. Finally, job preemption is allowed to manage higher priority submissions. To evaluate the effectiveness of the Hierarchical method, we compare its performance with an extension of the DP-based algorithm in (Saxena et al. 2020), that we suitably modified to adapt to our scenario. While our Hierarchical method considers the batch size of each job as a static parameter, fixed by the user upon submission as well as, e.g., the deadline, the DP-based method treats it as a dynamic parameter that can be adapted to guarantee better performance. To boost the DP-method results, we implemented multiple variants by relying on different proxy functions. Our comparison considers also a First-Principle method based on the Earliest-Deadline-First (EDF) policy which is used as a baseline also in other studies (Amaral et al. 2017).

Experimental results show that the Hierarchical method outperforms the other algorithms particularly when the system load is relatively high, achieving an average cost reduction up to 99% against the DP-based methods and 96% with respect to EDF. Additional tests highlighted that the DP-based algorithms obtain comparable results with respect to the Hierarchical method only when they can exploit a four times larger amount of resources, making the latter the most suitable in order to achieve good results employing a limited number of nodes. The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 describes the hierarchical framework proposed in (Filippini et al. 2020). Section 4 introduces the novel DP based algorithm. Section 5 presents the experimental setup and the results of comparing both the Hierarchical method against the DP methods and the First-Principle approach. Finally,

Section 6 draws conclusions and outlines future works.

2 Related work

Optimizing job scheduling and GPU resources management in a Deep Learning (DL) context is a popular problem in these years. GPUs lead to an unprecedented computing power, still hard to fully exploit (Steinberger 2018). Both robust theoretical frameworks and effective practical solutions are thus needed to support job scheduling (Tan et al. 2019).

The Hierarchical method proposed in (Filippini et al. 2020) represents one of the first attempts to tackle jointly the problems of online DL job scheduling and resource selection on multiple virtualized GPUs, while most of the existing literature is mainly focused on one of these aspects. Many works, e.g., (Amaral et al. 2017; Xiao et al. 2018; Bao, Peng, and Wu 2019; Chaudhary et al. 2020; Mahajan et al. 2020), rely on GPU requests submitted by the users and determine only the optimal job scheduling to optimize different objectives. Other proposals focus instead on the resource selection problem, delegating the scheduling to simple mechanisms as First-In-First-Out (FIFO) or EDF (Peng et al. 2018; Saxena et al. 2020; Peng et al. 2021; Yeung et al. 2022).

In particular, (Amaral et al. 2017) proposes a topology-aware scheduling policy for DL jobs in cloud environments, which provides a placement strategy able to satisfy workload requirements preventing also application interference. Gandiva (Xiao et al. 2018) is a scheduling framework that improves latency exploiting heterogeneity and recurrent behaviors of DL jobs while running mini-batch iterations. Harmony (Bao, Peng, and Wu 2019) deep reinforcement learning-based scheduler evaluates the impact of co-location to reduce interference, aiming at maximizing GPUs and nodes usage and reducing jobs completion time. $Gandiva_{fair}$ (Chaudhary et al. 2020) exploits a central, gang-aware scheduler for large jobs that span multiple servers, and a local, per-server, gang-aware scheduler for small jobs, in order to maximize inter-user fairness. Fairness is crucial also for Themis (Mahajan et al. 2020), 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.

Optimus (Peng et al. 2018) is a Kubernetes scheduler especially designed to manage DL jobs on a shared distributed containerized environment. It minimizes the training time by estimating job execution times through online resource-performance models and reduces communication overheads by placing jobs on the minimum number of servers that allow to deploy an equal amount of workers/parameter servers (PS). DL^2 (Peng et al. 2021) combines an offline supervised learning and an online reinforcement learning-based approach for resource selection, setting the number of workers/PS to adopt for DL training jobs. The work in (Yeung et al. 2022) proposes an interference-aware and prediction-based resource manager that evaluates the quality of placement decisions through GPU utilization. Finally, (Saxena et al. 2020), which is used as benchmark in this work, develops an optimization formulation where the optimal job batch size is set according to their scaling efficiency. Moreover, it proposes a dynamic programming-based heuristic algorithm to determine an effective resource allocation, while jobs are scheduled relying on a FIFO mechanism.

3 The Hierarchical method

This paper aims to analyze the performance of a resource selection and scheduling algorithm designed in a hierarchical framework (Filippini et al. 2020) for the management of DL training jobs on a GPU-based virtual machine cluster, comparing it with a novel Dynamic-Programming-based algorithm and a First-Principle method based on the EDF policy. The algorithm, initially proposed in (Filippini et al. 2020), is denoted in the following as Hierarchical method. This section summarizes its main features in order to make this paper self-contained and readily comprehensible. In the reference framework (see Figure 1), multiple jobs can run concurrently on the same node and each job can be allocated on a single node (featuring, possibly, multiple GPUs). As already mentioned, a deadline is associated to each job upon submission, and a penalty proportional to the job's priority is incurred in case this deadline is exceeded. Moreover, job preemption is allowed.

Incoming jobs are submitted to a central queue $\mathcal{J}$, that, according to a Round Robin (RR) policy, distributes them in local queues $\mathcal{J}_k$, each one managed by a local controller k. The set $\mathcal{N}$ of available nodes, which can be provisioned with a VM type v selected from a cloud provider's catalog $\mathcal{V}$, is partitioned among the K controllers so that each one considers N/K instances. Each VM type v available in the catalog is characterized by a set of available GPUs $\mathcal{G}_v = \{1, 2, \ldots, G_v\}$, and by a time unit cost c_v. Each job $j \in \mathcal{J}$ has a submission time, a deadline d_j, and a tardiness weight ω_j that is used to characterize its priority. Indeed, the penalty due to the violation of its deadline is defined as $\omega_j \tau_j$, where the tardiness τ_j is computed *a posteriori* as

$$\tau_j = \max\{0, T_j - d_j\}, \tag{1}$$

and T_j is the ending time of the execution of job j.

Each local controller k aims at: i) determining which jobs must be executed in the current scheduling step and which must be postponed, ii) choosing from the cloud provider catalog the best VM type to deploy on each node, iii) partitioning efficiently the available GPUs among the running jobs on each node.

Therefore, each local controller solves a joint Capacity Allocation (CA) and Jobs Scheduling (JS) problem in an online setting, every time a job is completed or a new job is submitted or after a certain time interval denoted by Δt if none of the two previous events happen. We assume that jobs execution can be stopped and resumed from a checkpoint to account for changes in the allocated resources.

The expected execution times of job j when it is deployed on a VM type v with g GPUs, denoted by t_{jvg}, are estimated through the machine learning models proposed in (Lattuada et al. 2022), with an average percentage error below 11%.

Figure 1 shows an example featuring six DL training jobs, sent to three local controllers so that, e.g., controller k manages a local queue $\mathcal{J}_k = \{j_2, j_5\}$. The N/K nodes in

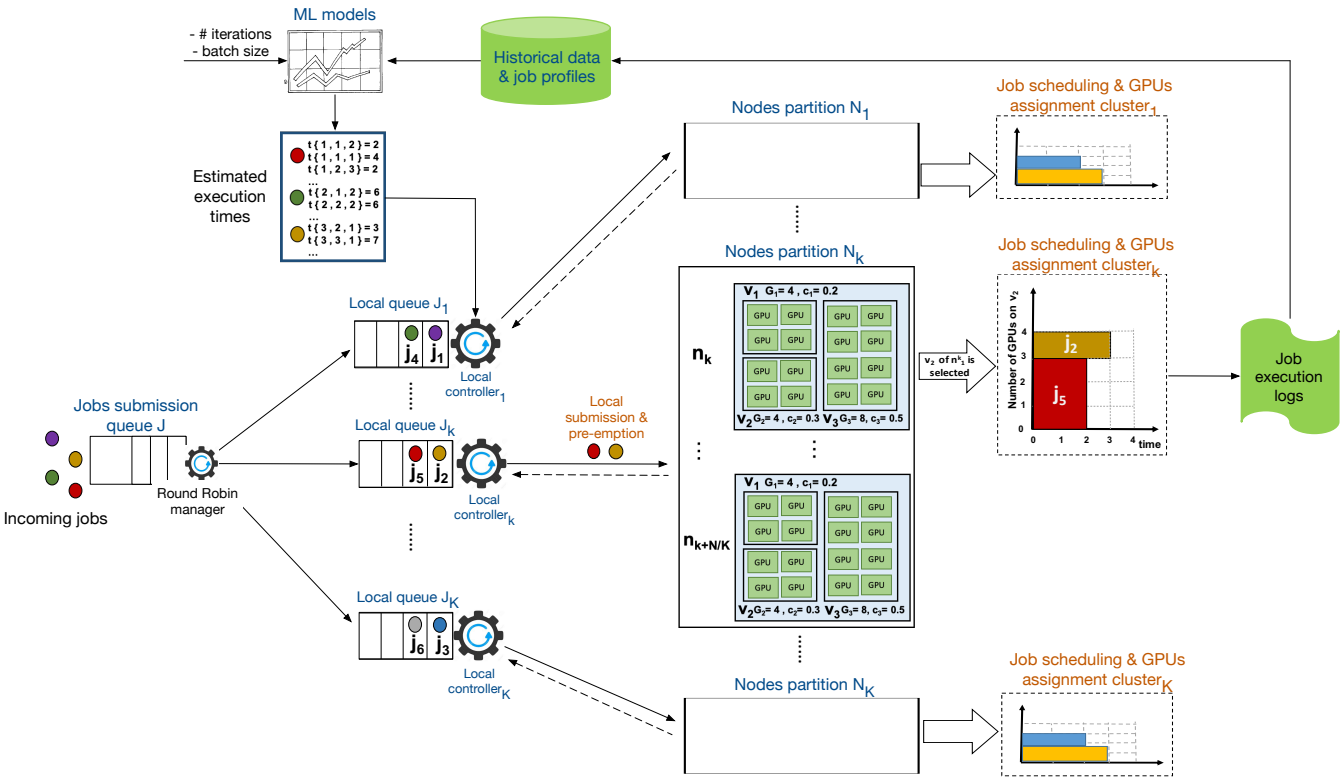

Figure 1: Reference framework.

$\mathcal{N}_k = \{n_k, \ldots, n_{k+N/K}\}$ can be configured with VMs of three types. Two types (v_1 and v_2) have four GPUs while v_3 has eight GPUs. Each VM type has its own time unit cost $(0.2, 0.3, 0.5)$ \$/h. Sample execution time estimates are reported in the blue box. The local controller k schedules jobs j_2 and j_5 to run on the same node n_1^k selecting the VM type v_2 , with 4 GPUs. Job j_5 will end first; if no other jobs are submitted in the while, the problem is solved again by the local controller: job j_2 can be preempted or can continue in the next time slot, possibly with a different configuration (VM type and/or number of GPUs). A similar procedure is performed when a new job joins the local queue.

Each local controller k solves a Mixed-Integer Linear Programming (MILP) problem, whose complete formulation is reported in (Filippini et al. 2020), to determine which VM type $v \in \mathcal{V}$ to select on each available node $n \in \mathcal{N}_k$ (through the binary variables y_{nv}^k), and the optimal deployment of job $j \in \mathcal{J}_k$ (through the binary variables x_{jnvg}^k, which are 1 if job j runs on node n with VM type v and g GPUs). Since a single VM type can be selected on each node, the set $\mathcal{N}_k$ is often referred to in the following as set of assignable VMs.

The list of all problem parameters and variables is reported in Table 1.

The objective function for controller k reads:

$$\min \sum_{j \in \mathcal{J}_k} \omega_j \left(\tau_j^k + \rho \widehat{\tau}_j^k\right) + \mu \sum_{\substack{n \in \mathcal{N}_k \\ v \in \mathcal{V}}} (G_v y_{nv}^k - \sum_{\substack{j \in \mathcal{J}_k \\ g \in \mathcal{G}_v}} g x_{jnvg}^k) + \sum_{\substack{j \in \mathcal{J}_k \\ n \in \mathcal{N}_k}} \alpha_{jn}^k \pi_{jn}^k \tag{2}$$

The first term represents the costs due to the accumulated tardiness. In particular, the worst-case tardiness $\widehat{\tau}_j^k$ is defined, for any job $j \in \mathcal{J}_k$, as:

$$\widehat{\tau}_j{}^k = \max\{0, T_c + \Delta t + \min_{v,g} t_{jvg} - d_j\}, \tag{3}$$

where T_c is the current time, Δt is the scheduling time interval and t_{jvg} is the expected execution time of job j on the VM type v using g GPUs. The worst-case tardiness aims to penalize postponed jobs, who may be resumed too close to the deadline, and assumes a different value for every scheduling step.

The second term in Equation (2) is proportional to the difference between the number of used GPUs and the number of available GPUs from each selected VM type, and so it penalizes idle resources (μ is a positive constant acting as a Lagrange multiplier).

Finally, the third term in Equation (2) corresponds to the sum over all nodes of the total execution costs of the first job that will complete on them. Here, α_{jn}^k is a binary variable that is equal to 1 if the job $j \in \mathcal{J}_k$ is the first ending job on node $n \in \mathcal{N}_k$, and 0 otherwise, while π_{jn}^k is the total execution cost of job $j \in \mathcal{J}_k$ on node $n \in \mathcal{N}_k$. Although binding the execution costs only to the first ending job might seem a short-sighted approach, the experiments conducted in (Filippini et al. 2020) and in Section 5.2 confirm its effectiveness. Finally, note that we are neglecting re-configuration costs of running nodes since this would require a few minutes while DL training jobs run for several hours (or days).

Table 1: Notation of the MILP model

Problem parameters	
$\mathcal{J}_k$	set of jobs submitted to the local queue k
$\mathcal{N}_k$	set of available nodes managed by the local controller k
G_v	number of available GPUs on the VM type v
d_j	deadline of job j
ω_j	tardiness weight of job j
t_{jvg}	execution time of job j when running on VM type v with g GPUs
Δt	scheduling time interval
μ	a penalty coefficient for unused GPUs
ρ	a penalty coefficient for postponed job
Local controller k variables	
y^k_{nv}	1 if VM type v is chosen on node n, 0 otherwise
x^k_{jnvg}	1 if job j is executed on node n with VM type v on g GPUs, 0 otherwise
τ^k_j	tardiness of job j
$\widehat{\tau}^k_j$	worst-case tardiness of job j if it is postponed
π^k_{jn}	execution cost of job j on node n
α^k_{jn}	1 if job j is the first-ending job on node n, 0 otherwise

4 A Dynamic Programming-Based Alternative Method

To assess the effectiveness of our approach, we compare its results with the outcomes of a novel extension of the DP-based method initially proposed in (Saxena et al. 2020).

The initial work introduced a resource allocation strategy for DL training jobs that leverages dynamic programming to determine, for each job, the number of GPUs to be allocated and its optimal batch size. The optimization is periodically performed, with a time period Δt. Note that the optimization method is developed in a centralized way, thus at any rescheduling point it considers the complete queue $\mathcal{J} = \bigcup_k \mathcal{J}_k$ and the set of nodes $\mathcal{N} = \bigcup_k \mathcal{N}_k$, consequently all the variables do not need the index of the local controller k. Furthermore, the DP algorithm has an additional decision variable: at each rescheduling point the job batch size can be selected to optimise the job expected execution time. Therefore, while the job execution times were denoted in Section 3 as t_{jvg}, here we define them as t_{jvgb}, where b is the optimal batch size selected for job j in the current scheduling step.

The core of our extension with respect to (Saxena et al. 2020) is to add the concept of VMs and to optimally select VMs size, which was not considered in the reference work. Indeed, the original method worked under the assumption that all nodes were physical machines hosting GPUs of homogeneous types. In contrast with our Hierarchical method (see Section 3), (Saxena et al. 2020) prescribes that a single job can be executed on each node. Since only one VM type is selected on each node, this is equivalent to prescribe that one job can run on each VM. This has a strong impact on the choices the algorithm can make concerning the resource allocation. First of all, since any VM type v can be selected from the catalogue $\mathcal{V}$ to run on any node $n \in \mathcal{N}$, the VM type and the number of GPUs assigned to a job $j \in \mathcal{J}$ do not influence the resources assigned to all the other jobs: the only constraint is given by the total number of available nodes. Moreover, all the GPUs available in a VM are used to run the job deployed on it, since they cannot be assigned to other jobs and idle resources determine higher operational costs.

Our extension, that in the following will be named *DP algorithm*, is based on the *Optimal Substructure property*, that in our context can be stated as follows:

Property 1 (Optimal Substructure) *Consider the cost function $\mathcal{F}$ and let $\mathcal{F}_{opt}\left(\{j_i\}_{i=1}^{J}, N\right)$ be its optimal value for jobs $\{j_i\}_{i=1}^{J}$ when the number of available nodes is N. We say that the problem has an Optimal Substructure if:*

$$\mathcal{F}_{opt}(\{j_i\}_{i=1}^{J}, N) = \min_{n \in \{0,1\}} \mathcal{F}_{opt}(\{j_i\}_{i=1}^{J-1}, N-n) + \mathcal{F}_{opt}(j_J, n)$$

Property (1) can be interpreted as follows: given an optimal solution, if we consider as a collection of sub-problems any partition of the jobs and the nodes on which they run, the solution is optimal also for the single sub-problems. Moreover, also the converse is true: the partition of nodes is such that the sum of the costs provided by the optimal solutions of the single sub-problems is optimal for the original problem.

The general structure of our DP approach is reported in Algorithm 1. At lines 2 and 3, three parameters are initialized: J denotes the number of submitted jobs that can be executed in the current scheduling step, i.e. the length of the queue $\mathcal{J}$. F is a $J \times N$ table such that $F(i, n)$ is the cost associated to the optimal solution considering only jobs $\{j_l\}_{l=1}^{i}$ and n nodes. Finally, SOL is a $J \times N$ table such that SOL(i, n) stores the setup selected to execute job j_i in order to achieve the optimal cost $F(i, n)$. Before entering in the first loop, the first row of F is initialized to 0 since no operations are required if there are no jobs. Then, in lines 4-7, the optimal configuration is selected for every job according to the value of a proxy function $\mathcal{F}_1$ (defined in the following), and both 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 $\mathcal{F}_2$. Lines 8-18 aim to find the optimal cost and resource allocation for jobs $\{j_l\}_{l=1}^{i}$ considering only h nodes, exploiting Property 1. Finally, at lines 21-27 we allocate resources to all jobs exploiting a backward scheme.

To define the proxy functions $\mathcal{F}_1$ and $\mathcal{F}_2$, we consider the binary variables y_{nv}, equal to 1 if the VM type $v \in \mathcal{V}$ is selected on node $n \in \mathcal{N}$ (see Section 3), and z_{jn}. These were used in the complete formulation proposed in (Filippini et al. 2020), and are equal to 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. The new problem parameters and variables are summarized in Table 2.

Starting from the problem definition in Section 3, we introduce a first proxy function as:

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

The first term represents the penalty for deadline violations, the second one measures the operational costs related

Algorithm 1 General structure of the DP algorithm

Result: Optimal setup for each submitted job given the jobs queue $\mathcal{J}$ and N nodes
$J \leftarrow LENGTH(\mathcal{J})$, $F \leftarrow +\infty_{J \times N}$
$SOL \leftarrow 0_{J \times N}$, $F(0,:) \leftarrow 0$
for $i = 1, \ldots, J$ **do**
  find the best set up C_i for j_i w.r.t. $\mathcal{F}_1$
  $c_{assign} \leftarrow \mathcal{F}_2(j_i, C_i)$
  $c_{queue} \leftarrow \mathcal{F}_2(j_i, \{\})$
  for $h = 1, \ldots, N$ **do**
    $p_{assign} = c_{assign} + F\left(\{j_m\}_{m=1}^{i-1}, h-1\right)$
    $p_{queue} = c_{queue} + F\left(\{j_m\}_{m=1}^{i-1}, h\right)$
    if $p_{assign} < p_{queue}$ **then**
      $F\left(\{j_m\}_{m=1}^{i}, h\right) = p_{assign}$
      $SOL(i,h) = C_i$
    else
      $F\left(\{j_m\}_{m=1}^{i}, h\right) = p_{queue}$
      $SOL(i,h) = \{\}$
    end if
  end for
end for
$i \leftarrow J$ $k \leftarrow K$
while $i > 0$ **do**
  set the optimal set up for j_i $C_i = SOL(i,k)$
  $i \leftarrow i - 1$
  if $C_i \neq \{\}$ **then**
    $k \leftarrow k - 1$
  end if
end while

to VMs usage, and the third one is used to penalize the postponement of jobs via the worst-case tardiness $\hat{\tau}_j$ defined in Equation (3). In the following, we will denote the method characterized by $\mathcal{F}_1 = \mathcal{F}_2 = \mathcal{F}_{WCT}$ as *DP(WCT)*.

In order to enhance the results obtained by DP(WCT), we tried to develop alternative proxy functions. 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 shown in the next sections). This choice was coupled with two alternatives for $\mathcal{F}_2$. In the first setting, denoted as *DP(FastWCT)*, we kept $\mathcal{F}_2 = \mathcal{F}_{WCT}$. In the second scenario, denoted instead as *DP(FastB)*, we defined $\mathcal{F}_B$ by replacing the worst-case tardiness $\hat{\tau}_j$ with a positive constant B. Indeed, if the deadline 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 would negatively affect the performance in the long term, since postponed jobs risk to violate their deadlines if, due to the arrival of new jobs, 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:

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

The first term of $\tilde{\mathcal{F}}$ is obtained by substituting the tardiness τ_j with an *adjusted tardiness* defined as:

$$\tilde{\tau}_j = \max\{0, T_c + t_{jvgb} - d_j\}, \quad (6)$$

which measures the delay of job j with respect to its deadline if it is fully executed with the current configuration (i.e., assuming that no migration occurs in the following scheduling steps). Due to this definition, it 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 as *DP(AdjWCT)*.

Note that, for a fair comparison among different DP-based algorithms, we always compute the final costs with the same proxy function, namely $\mathcal{F}_{WCT}$, regardless of the selected proxy functions $\mathcal{F}_1$ and $\mathcal{F}_2$.

We performed different experiments with the aforementioned methods, varying the time elapsed between two scheduling steps, Δt. For the sake of space, we discuss in the following section only the results obtained with $\Delta t = 15\ min$, which guaranteed the best trade-off between results quality and rescheduling frequency, comparing them with our method.

Table 2: Notation of the DP algorithms

Problem parameters	
$\mathcal{J}$	set of submitted jobs
$\mathcal{N}$	set of available nodes
c_v	time unit cost of the VM type v
d_j	deadline of job j
ω_j	tardiness weight of job j
t_{jvgb}	execution time of job j when running on VM type v with g GPUs and batchsize b
Δt	scheduling time interval
Problem variables	
y_{nv}	1 if VM type v is chosen on node n, 0 otherwise
z_{jn}	1 if job j is executed on node n, 0 otherwise
r_j	1 if job j is postponed, 0 otherwise
τ_j	tardiness of job j
$\hat{\tau}_j$	worst-case tardiness of job j if it is postponed
$\tilde{\tau}_j$	adjusted tardiness of job j

5 Experimental results

In this section we report the result of the comparison of the Hierarchical method with three of the DP algorithms presented in Section 4: DP(WCT), DP(FastWCT), and DP(AdjWCT). We omit the results of DP(FastB), since overall it shows a behaviour very close to DP(FastWCT). We considered a very large set of representative scenarios, randomly generated as described in Section 5.1. The results obtained by the Hierarchical method and the DP algorithm variants are compared in Section 5.2, drawing conclusions about their efficiency in relation to the amount of employed resources.

The results with all methods were collected exploiting a Ubuntu 18.04 VM based on a dual Intel Xeon Silver 4114 CPU at 2.20GHz with overall 40 cores and 64GB of memory. The largest problem instance (100 nodes and 1,000 jobs) can be solved by the Hierarchical Method (with Gurobi 9.0 exploiting all cores) in less than one minute on average (the computation time of the k-th controller is around $40 - 50s$,

see (Filippini et al. 2020) for an in depth analysis). The average time required to run the alternative methods to execute instances of different size are reported in Table 3.

The datasets and the source code employed to perform the experiments reported in this paper are available at https://zenodo.org/record/6591444#.YpPBlDlBxH4.

Table 3: Computational times (in seconds) required by the DP-based and the Hierarchical methods

N° nodes	Hierarchical	WCT	FastWCT	AdjWCT
10	40.724	6.963	2.956	3.355
20	41.475	19.645	6.888	8.078
30	42.254	29.740	11.158	13.230
40	43.062	38.998	15.124	17.538
50	43.902	59.700	20.710	24.530
60	44.776	74.620	24.820	29.553
70	45.685	94.380	30.120	36.088
80	46.632	116.528	36.164	43.388
90	47.619	142.298	43.571	52.225
100	48.649	164.885	50.413	59.855

5.1 Experimental setup

As representatives of long-running Deep Learning (DL) training jobs, we selected different neural networks (i.e., Alexnet, Resnet, VGG, and DeepSpeech) implemented with PyTorch and Tensorflow frameworks. They are significantly heterogeneous in terms of resource usage: VGG performance is heavily related to the available computational power, while Alexnet and DeepSpeech performance are mainly determined by disk-access efficiency and by the GPU memory size and speed. Finally, Resnet revealed to be characterized by a balanced type of workload, for additional details see (Lattuada et al. 2022). For each network-framework pair, several application instances have been created by varying the epochs number.

Table 4: 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

The considered VM catalog (reported in Table 4) is composed of 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 hypothetical 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.

To verify the effectiveness and generality of the proposed approach, several random problem instances were generated

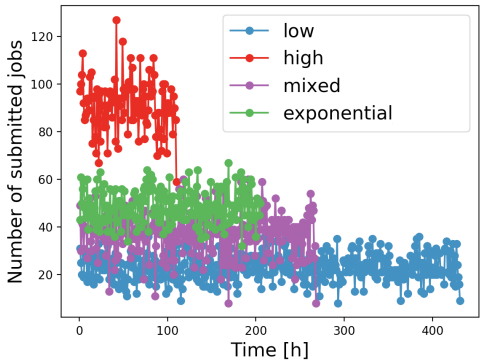

Figure 2: Job submissions under different workloads

using the parameters described in the following.

We varied the number N of available nodes in the cluster from 10 to 100. The number of submitted jobs in each instance is set to $J = 10N$.

The number of controllers K for the Hierarchical method has been set to $N/5$, i.e., each local controller has to manage 5 nodes (or VMs).

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., (Amaral et al. 2017)), 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 other instances, inter-arrival times were generated as described in (Saxena et al. 2020). Arrivals are sampled from a Poisson distribution, considering three possible rates. Let λ 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 $\varepsilon\ n_{max}\ \lambda$, while the *low* rate is $\varepsilon\ n_{max}\ \lambda/4$. We defined n_{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 ε to match the peak load of the system to real-life scenarios reported in (Peng et al. 2021), 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 (Saxena et al. 2020)).

The distributions of jobs arrivals for a scenario featuring $N = 1000$ and $J = 10000$ are reported in Figure 2. We used the aforementioned traces of jobs 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 involve the execution costs depending on the chosen VMs and the tardiness costs of jobs that complete their execution after the deadline.

For each value of the cluster size and each arrival rate, three problem instances were built by changing the seed of the random distribution.

The remaining parameters are set as follows. As stated previously, the periodic scheduling time interval Δt is set to 15 min. The deadline d_j for each job is randomly generated according to a uniform distribution in the range $[\min_{vg}\{t_{jvg}\},\ 3\min_{vg}\{t_{jvg}\}]$ for all the methods, independently on the selected batch size. This guarantees fair comparisons, since the batch size is a dynamic parameter for the DP methods.

The tardiness weights ω_j are randomly generated in the interval $[0.003, 0.015]$ \$/hour with a uniform distribution. In this way, for any job whose deadline is violated, the average time unit delay penalty is almost ten times larger than the time unit execution cost. Concerning the objective function adopted by the Hierarchical method, the postponed job penalty ρ is set to 100 while the μ parameter is set equal to 1 (given the objective function adopted in the problem formulation, any positive value forces the use of all available GPUs).

The results of both the Hierachical method (hereafter denoted as HM) and the three DP-based algorithms of Section 4 have been compared, as in other literature proposals (see, e.g., (Amaral et al. 2017)), against those obtained with the Earliest-Deadline-First method. In particular:

- For each problem instance characterized by N available nodes and J jobs generated as described in Section 5.1, different solutions were determined by exploiting all methods.
- Then, the same jobs traces were considered by all methods except HM, progressively increasing the amount of available resources by exploiting $2N$, $4N$ and $8N$ nodes. The results associated to the $8N$ and $4N$ scenarios were found to be very similar, since the resources available in a system with $4N$ nodes were already enough to run all jobs concurrently. Thus, the $8N$ results are not reported here.

The purpose of this set of experiments is twofold: first, we want to quantify the relationship between the performance of HM and those of all the other methods, and check if this depends on the instance size (number of nodes and jobs). Second, since HM is the only method which allows to execute multiple jobs on a single node, the first comparison might not be fair. So, in order to balance that constraint, we also compare the results obtained by HM with those obtained by the other methods employing more nodes.

All HM solutions were obtained by setting in Gurobi the mixed-integer programming gap (i.e., the difference between the current upper and lower bounds of the MILP solver) to 5%. EDF and the DP-based methods are implemented in C++.

5.2 Comparative analysis

For the sake of space, we report here the results for the exponential, low and high distributions. The results of the simulations for the mixed distribution were found to have a trend very similar to the high distribution. For each considered scenario, the first row of Figure 3 shows the ratio between the average total cost obtained with all proposed methods and the average total cost of EDF. The plots in the left column represent the outcome associated to the exponential distribution, while the high distribution is represented by the plots in the middle and we finally we find the low distribution in the right column. We observe in the first row of Figure 3 that HM yields the best performance when N nodes are exploited by all five algorithms for all the distributions. The average cost reductions are between:

- 16.48% (low rate) and 95.53% (high rate) compared to EDF
- 47.85% (low rate) and 93.97% (high rate) compared to DP(FastWCT)
- 35.96% (low rate) and 95.50% (high rate) compared to DP(AdjWCT)
- 95.36% (low rate) and 98.70% (high rate) compared to DP(WCT).

For all the distributions, DP(WCT) is the method which leads to the worst performance. This is due to the fact that, as specified in the previous section, the worst case tardiness τ_j defined in Equation (3) does not penalize jobs preemption properly. For the exponential and high distributions, DP(FastWCT) performs better than EDF and the other DP-based algorithms. This is not surprising: indeed DP(FastWC) is designed for situations with a high system load. This is not valid for the low distribution, in which DP(AdjWCT) performs better than DP(FastWCT) and EDF performs better than all the DP-based algorithms, with an overall cost comparable even to the one obtained by HM. Indeed, in the low rate scenario the system load is reduced, so there is no need to look for the fastest setup; consequently, it is easier to meet the deadlines even with simple algorithms like EDF.

Concerning the set of experiments exploiting a larger number of nodes, first we note that, regardless the number of available nodes, DP(WCT) always yields the worst results for all distributions. Moreover, as the number of available nodes increases, DP(AdjWCT) and EDF perform better than DP(FastWCT): indeed we are decreasing the ratio J/N, i.e., the system load is reducing. It is interesting to see that, when $4N$ (and also $8N$) nodes are considered, EDF and DP(AdjWCT) achieve comparable (or, sometimes, marginally better) results with respect to those obtained with HM (for which only N nodes are exploited). The maximum gain w.r.t. HM is reached by DP(AdjWCT) for the high distribution when the number of nodes is equal to $4N$ (and $8N$), and it is less then 20%. However, exploiting a higher number of nodes could be financially not feasible, especially when relying on reserved instances. Indeed, even if there is the possibility to have a reduced hourly cost for resources, this often can be done through an additional yearly cost. Therefore, methods that yield to equivalent costs requiring less resources are preferred.

6 Concluding remarks

This paper compares the performance of the Hierarchical method originally presented in (Filippini et al. 2020), for the online joint capacity planning and jobs scheduling for

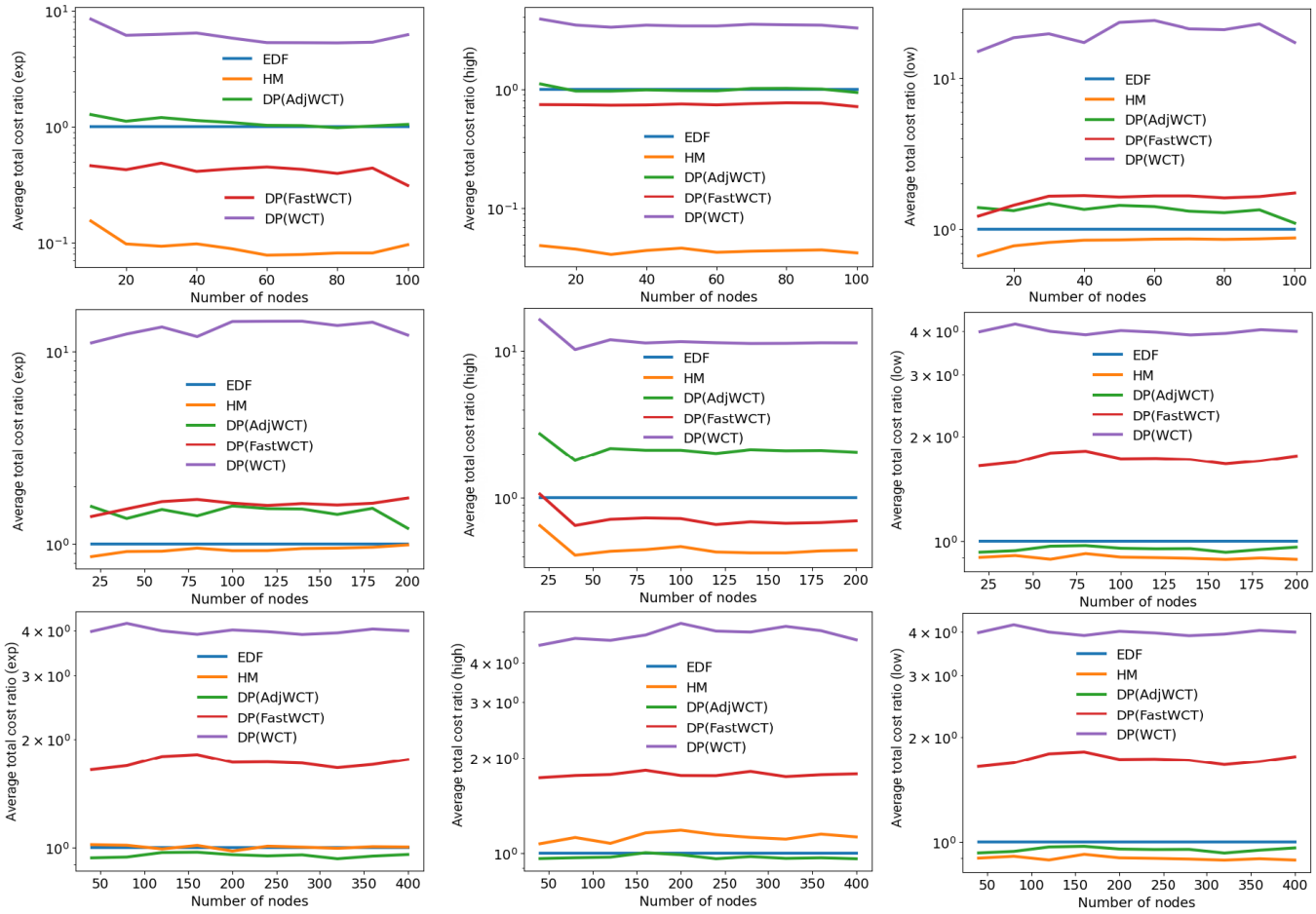

Figure 3: Total costs for different number of nodes and distribution of jobs submission. The horizontal axis represents the number of nodes employed, while the vertical axis represents the ratio between the cost obtained with a fixed method and the one obtained by EDF under the same conditions

DL training in cloud deployments, against three Dynamic-Programming (DP)-based methods adapted from (Saxena et al. 2020), all sharing the same structure but with different proxy functions. The results of the presented experimental campaign show how the Hierarchical method achieves very significant costs savings with respect to Earliest-Deadline-First (EDF) and the DP-based methods for an exponential inter-arrival and high rate of incoming jobs, while for a lower rate the gain is less remarkable but still relevant. In particular, the Hierarchical method achieves an average percentage cost reduction between 16% and 96% with respect to EDF, and between 36% and 99% compared to the DP-based algorithms.

Among the compared methods, the Hierarchical approach is the only one that allows the execution of multiple jobs on the same node. Therefore, to guarantee a fair comparison, we expanded the experiments by comparing its results with those obtained with the other methods when they consider the same jobs traces but a larger number of nodes. We observed that, in the best case, this leads to a gain less than 20% with respect to the Hierarchical approach, which does not worth the usage of so many resources.

In our research agenda we plan to integrate the Hierarchical method with a jobs queue manager and to validate the results achieved in an industry setting.

Acknowledgments

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

References

Amaral et al. 2017. Topology-aware gpu scheduling for learning workloads in cloud environments. In *HPCNSA Proc.* ACM.

Bao, Y.; Peng, Y.; and Wu, C. 2019. Deep learning-based job placement in distributed machine learning clusters. In *IEEE INFOCOM 2019*, 505–513.

Cao et al. 2019. Gpu-accelerated feature tracking for 3d reconstruction. *OLT* 110:165–175.

Chaudhary et al. 2020. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *EUROSYS*.

Filippini et al. 2020. Hierarchical scheduling in on-demand gpu-as-a-service systems. In *SYNASC2020*, 125–132.

Lattuada et al. 2022. Performance prediction of deep learning applications training in gpu as a service systems. *Cluster Computing* 25:1279–1302.

Mahajan et al. 2020. Themis: Fair and efficient GPU cluster scheduling. In *USENIX (NSDI 20)*, 289–304.

Peng et al. 2018. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*.

Peng et al. 2021. Dl2: A deep learning-driven scheduler for deep learning clusters. *IEEE TPDS* 32(08):1947–1960.

Ranjbarzadeh et al. 2021. Brain tumor segmentation based on deep learning and an attention mechanism using mri multi-modalities brain images.

Saxena et al. 2020. Effective elastic scaling of deep learning workloads. In *MASCOTS*, 1–8.

Steinberger, M. 2018. On dynamic scheduling for the gpu and its applications in computer graphics and beyond. *IEEE CGA* 38(3):119–130.

Tan et al. 2019. A virtual multi-channel gpu fair scheduling method for virtual machines. *IEEE TPDS* 30(2):257–270.

Wang et al. 2019. I-vector features and deep neural network modeling for language recognition. *CS Proc.* 147:36–43.

Xiao et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *USENIX OSDI*.

Yeung et al. 2022. Horus: Interference-aware and prediction-based scheduling in deep learning systems. *IEEE TPDS* 33(1):88–100.