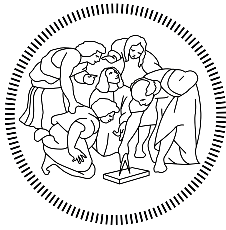

Analysis of Scheduling Algorithms for Deep Learning Training Jobs running on Virtualized GPU-based Clusters

M.Sc. Thesis of
Luel Mieraf Kiros

Supervisor:
Prof. Danilo Ardagna

Co-supervisors:
Federica Filippini

Programme:
COMPUTER SCIENCE AND ENGINEERING

DIPARTIMENTO DI INGEGNERIA INFORMATICA POLO TERRITORIALE DI COMO
Politecnico di Milano

Abstract

Machine Learning and Deep Learning are the topics which are paid huge attention since the last several years. While Machine Learning utilizes simpler ideas, deep learning works with artificial neural networks, which are designed to mirror how people think and learn. The recent adoption of GPUs as parallel general purpose processors partially satisfied this need, but the high costs associated with this technology, even in the cloud, dictate the need to design efficient capacity planning and job scheduling algorithms to reduce operational costs through resource sharing.

The proposed work tackles the problems of capacity planning and job scheduling together. In the envisioned scenario, the complexity of the problem, which poses a major challenge in terms of modeling and solvability, is compounded by the fact that capacity allocation and scheduling are evaluated in an online environment. Deep Learning training jobs are submitted in a continuous fashion, so that no scheme can be detected in their arrival times or features, especially as regards priority.

Inspired by greedy and local search techniques, heuristic methods have been developed to form efficient and scalable solutions to the proposed problem.

An experimental campaign proves the feasibility of the approaches developed for practical scenarios, showing considerable improvements in the computational time needed to determine solutions of good quality.

keywords: Machine Learning, Deep Learning, Heuristic Methods, GPU

Sommario

Machine Learning e Deep Learning sono gli argomenti a cui è stata prestata grande attenzione negli ultimi anni. Mentre l'apprendimento automatico utilizza idee più semplici, il Deep Learning funziona con reti neurali artificiali, progettate per rispecchiare il modo in cui le persone pensano e apprendono. La recente adozione delle GPU come processori paralleli per scopi generici ha parzialmente soddisfatto questa esigenza, ma i costi elevati associati a questa tecnologia, anche nel cloud, impongono la necessità di progettare efficienti algoritmi di allocazione della capacità e scheduling, per ridurre i costi operativi attraverso la condivisione delle risorse.

Il lavoro proposto affronta insieme i problemi della allocazione della capacità e scheduling. Nello scenario immaginato, la complessità del problema, che rappresenta una grande sfida in termini di modellizzazione e soluzione, è aggravata dal fatto che l'allocazione della capacità e lo scheduling sono valutati in un modo online. I lavori di training per il Deep Learning vengono inviati in modo continuo, in modo che non sia possibile rilevare alcun pattern nei tempi o nelle caratteristiche di arrivo, in particolare per quanto riguarda la priorità.

Ispirati da tecniche di ricerca greedy e di local search, sono stati sviluppati metodi euristici per offrire soluzioni efficienti e scalabili al problema proposto.

Una campagna sperimentale dimostra la fattibilità degli approcci sviluppati per scenari pratici, mostrando notevoli miglioramenti nel tempo di calcolo necessario per determinare soluzioni di buona qualità.

List of Figures

2.1 CLOUD COMPUTING ARCHITECTURE 16

2.2 CLOUD COMPUTING MODELS 17

2.3 GENERAL GREEDY ALGORITHM 21

3.1 REFERENCE FRAMEWORK 25

3.2 GREEDY ALGORITHM 27

3.3 SELECTION OF BEST NODE(RANDOM GREEDY) 32

4.1 GLOBAL STRUCTURE OF THE SOLUTION PROCESS . . 41

4.2 Comparison between Hierarchical Model and Random Greedy Algorithm 45

4.3 PERCENTAGE GAIN w.r.t EDF 46

4.4 COMPARISON BETWEEN CENTRALIZED GREEDY AND CENTRALIZED RANDOM GREEDY 47

4.5 (a) COMPARISON BETWEEN CENTRALIZED GREEDY AND HIERARCHICAL Random Greedy Algorithms 48

4.6 (b) COMPARISON BETWEEN CENTRALIZED GREEDY AND HIERARCHICAL 49

List of Tables

3.1 PROBLEM PARAMETERS 24

4.1 CHARACTERISTICS OF THE TARGET NODES 40

Acknowledgement

It has been a journey like no other, the experiences I had are there for me to treasure throughout my life. First, I would like to convey my deep gratitude to my supervisor Prof. DANILO ARDAGNA, and Co-supervisors FEDERICA FILIPPINI the Department of Electronics, Information and Bioengineering, Politecnico Di Milano for being the guiding light for me throughout the work. I was obliged to avail his insight and knowledge about all the various doubts and questions that I had throughout the work.

I would like to take the opportunity to thank all my friends and my family for believing in me and being there always to inspire and encourage me to achieve my goals in life.

Contents

1 Introduction **9**

2 State of the art **13**

2.1 Machine learning 13

2.2 Deep learning 13

2.3 Cloud computing 14

2.3.1 Cloud computing architecture 15

2.3.2 Deployment models 16

2.3.3 Cloud services 17

2.4 Virtual Machine 18

2.4.1 VM allocation problems 19

2.5 Heuristic approaches 20

2.5.1 Greedy algorithm 20

2.5.2 Randomized greedy algorithms 21

2.5.3 Local search 22

3 Methods **24**

3.1 Scalable methods for the joint capacity allocation and scheduling of DL jobs 24

3.1.1 Centralized Model 24

3.1.2 Hierarchical approach 25

3.1.3 Greedy algorithm 26

3.1.4 Randomized Greedy algorithm 31

3.1.5 Local Search 33

4 Result and Simulation **38**

4.1 Experiment Setting 38

4.2 Simulation 39

4.3 Comparison with first principle methods 43

4.3.1 Heuristic Approach 44
4.3.2 Hierarchical approach and Random Greedy Algorithm 44
4.3.3 Pure Greedy and Randomized Greedy Algorithms . . 45
4.3.4 Comparison between Centralized Greedy and Centralized Random Greedy Algorithms 46
4.3.5 Comparison between Centralized Greedy and Hierarchical Random Greedy Algorithms 47
4.4 Discussion 49

5 Conclusion and future work 51

Bibliography 53

Chapter 1

Introduction

Machine Learning and Deep Learning are the topics which have been paid huge attention since the last several years. This is very interesting in the sense that research in artificial intelligence started back in the mid 20th century with bold promises which were not materialized by the end of the century. Machine Learning systems are used to identify objects in images, transcribe speech into text, match news items, posts or products with users' interests, and select relevant results of a search. Increasingly, these applications make use of a class of techniques called deep learning.

Deep learning allows computational models that are made out of different preparing layers to learn portrayals of data with various degrees of abstraction. These strategies have drastically improved in classification problems like visual object recognition, object identification, and numerous different areas, for example, drug discovery and genomics.

Deep learning finds a perplexing structure in huge data collections by utilizing the back-propagation algorithm to demonstrate how a machine should change its inner parameters that are utilized to register the portrayal in each layer from the portrayal in the past layer. Deep convolutional nets have realized advancements in handling pictures, video, discourse, and sound, while intermittent nets have shone a light on successive information, for example, text and speech [9].

To empower deep learning algorithms to handle progressively complex issues, a savage power arrangement, scaling up to the framework level, is regularly applied. The quantity of utilized neurons becomes so as to manage an expanding measure of information, making the training procedure of the subsequent system increasingly more computationally demanding. The expansion in the necessary computational force expected to take care of these issues has been looked at in different manners [11].

The measure of the computational power required for ring DL applications, other than the need for high stockpiling abilities to manage massive or complex datasets, remains a central issue in the field of Machine Learning. The possibility of utilizing Graphics Processing Units as General Purpose equal processors has been investigated in the most recent years as a strategy ready to upgrade the computational power. The subsequent huge parallelism

empowers us to understand, with a sensible computational time and exertion, progressively complex undertakings, as those emerging in the fields of DL and Artificial Intelligence.

GPU speeding up, and particularly the chance of productively performing grid increases in equal, on account of deeply specific direct polynomial math libraries, is especially fit to DL training jobs, ensuring, as for CPU-based frameworks, speedup from 5 to 40x GPU-based frameworks, exploiting their design [3].

The way toward training deep learning applications remains a computationally serious assignment. Also, GPU-based servers are described by extensively significant expenses. As an outcome, they remain frequently unreasonably expensive for the overall population, comprising additionally of little associations with restricted financial plans.

Developing interest and the problems identified with the openness of GPU-based engineering decided, in the most recent years, an ensuing movement of Cloud administrations and arrangements intending to upgrade the utilization of those assets in various settings. The chance of approaching GPU-based frameworks as indicated by pay-to-go evaluating models has added to the across the board of those advancements applied to the arrangement of various issues, including the DL training issues previously mentioned.

The Cloud Computing worldview and the ensuing chance of approaching a preferably boundless computational and capacity power, the time unit expenses of Virtual Machines dependent on GPUs are still surprisingly high, being 5-8x more costly than those of VMs misusing just CPUs [3]. As an outcome, the issue of deciding proficient planning for deep learning training jobs and different applications that ought to be conveyed on those frameworks has still an incredible significance.

The joint issue is tended to in the writing by misusing various methodologies and it speaks to an extraordinary test as far as demonstrating and resolvability. The unpredictability is even exacerbated, in the analysed situation by the fact that capacity allocation and scheduling are investigated in an online setting, where numerous deep learning training jobs are submitted in a nonstop manner, so that no plan can be identified in their submission times

or qualities, especially regarding need. Besides, jobs can be preempted, in order to be able to prioritize, if needed, subsequently arriving jobs with different characteristics in terms of execution times and expected deadlines.

The proposed work intends to improve the adaptability of the arrangement by misusing an alternate Mixed Integer Linear Programming detailing, applied to the previously mentioned issue in a various leveled design. In addition, diverse heuristic methodologies, motivated by greedy and local search, have been planned and executed to decide a productive arrangement both as far as computational time and precision.

The expanding utilization of Cloud-based answers for algorithms that require a huge computational power and a lot of resources determines a likewise expanding interest for VM allocation and job scheduling problems.

Job scheduling issues are normally characterized as the portrayal of strategies that allow disseminating and resource spending plans among various tasks. A financial plan can consist of time or various types of resources, including, for example, computational power or capacity.

The circumstance where various applications ought to be simultaneously run sharing a fixed amount of resources regularly emerges in Deep Learning. The improvement of a DL program is typically partitioned into two primary steps: in the first, the program is composed and tried on a neighborhood machine or on a test cluster; in the subsequent one, the application is at long last sent and run on the production cluster.

The time expected to run a Deep Learning program in the primary stage can be extensively different as for the subsequent one. Without a doubt, the creating procedure normally requires to play out a few tests, including little datasets, hence it is less demanding for what concerns time and computational power. The fundamental issue that emerges in this specific situation, while considering GPUs use, is the fact that the iterative procedure of adjusting the code and afterward running a few little tests is probably going to leave resources idle for a lot of time.

The thesis is structured as follows: Chapter 2 discusses the state of the art, providing a general overview of the technical fields discussed in this

work and providing a perspective on the optimization approaches adopted for the design of the algorithms proposed. Chapter 3 discusses the various algorithms designed to evaluate an efficient and scalable solution. Chapter 4 presents the findings of some preliminary experimental research conducted in order to test the models and approaches discussed in this study. Lastly, Chapter 5 draws the findings from this thesis work and provides several potential directions for future study.

Chapter 2

State of the art

2.1 Machine learning

Machine Learning is one of the domains of Artificial Intelligence (AI). The objective of Machine learning is to understand the structure of data and find hidden correlation/patterns and fit that data into models that can be understood and utilized by people. Even though Machine Learning is a domain within computer science, it varies from traditional computational approaches. In traditional computing, algorithms are composed of programming instructions to calculate or solve a problem. In Machine Learning algorithms, a computer is trained on data inputs, and then it applies statistical analysis to the output values, checking if they fall within a specific range. This characteristic of machine learning enables computers to build models from sample data and automate decision making processes based on data input.

Currently, any user of technology has in one way or another benefited from Machine Learning, Facial recognition technology has allowed media platforms to automatically tagging images. Optical Character Recognition (OCR) technology has helped convert images to editable text. Biggest of all in the consumer side, it has helped big companies like Amazon, YouTube, Google, Spotify by using recommendation engines powered by Machine Learning to recommend movies, music and products more precisely. Autonomous vehicles' navigation is made possible by Machine Learning.

2.2 Deep learning

Deep Learning can be considered as a subset of Machine Learning. It is a field that depends on learning and enhancing its own by inspecting computer algorithms. While Machine Learning utilizes simpler ideas, Deep Learning works with artificial neural networks, which are designed to mirror how people think and learn. Up to this point, neural systems were restricted by computing power and along these lines were constrained in multifaceted nature. Deep learning has helped image classification, language interpretation, discourse acknowledgment. It tends to be utilized to take care of any pattern recognition issue and without human intervention.

Deep Learning is characterized by the attempt to generate and improve

knowledge by extracting patterns from raw data. This procedure, known as the learning process, can only be performed profitably in the presence of large amounts of data which can be analyzed using the algorithm chosen.

Artificial neural systems, including numerous layers, drive deep learning. Deep Neural Networks (DNNs) are such sorts of systems where each layer can perform complex tasks, for example, representation and reflection that comprehend images, sound, and content. Considered the quickest developing field in AI, Deep Learning represents a really troublesome advanced innovation, and it is being utilized by progressively more organizations to make new plans of action [11].

Neural systems are involved in layers of nodes, much like the human mind is composed of neurons. Nodes inside individual layers are associated with neighboring layers. The network is said to be deeper depending on the quantity of layers it has. A single neuron in the human mind gets a huge number of signals from other neurons. In an artificial neural system, signals travel among nodes and dole out relating loads. A heavier weighted node will apply more impact on the following layer of nodes. The last layer orders the weighted contributions to create a yield. Deep Learning frameworks require amazing equipment since they have a lot of information being prepared and include a few complex scientific computations. Indeed, even with such advanced hardware, however, Deep Learning training computations can take weeks.

Deep Learning systems require a huge amount of data to return precise outcomes; in like manner, data is taken care of as gigantic data collections. When handling the data, ANN can arrange data with the appropriate responses obtained from a progression of binary true or false inquiries including exceptionally complex numerical computations[11].

2.3 Cloud computing

The Cloud Computing worldview has been created to answer such a developing interest by giving resources that can be rented by clients in an on-request design. The principle explanation behind the presence of distributed computing is to diminish IT overhead for End clients and furthermore decrease the aggregate of On-Demand administrations and numerous different things. Distributed computing utilizes advances, for example, virtualization and utility-based valuing to meet the innovative and conservative prerequisites

of the client's interest in IT [7].

Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services. In addition, cautious scope organizations so as to purchase the necessary assets are not required, since Cloud clients can adjust the quantity of dispensed assets as indicated by their outstanding task at hand request in no time flat. This has the benefit of lessening the working expenses since inert resources can typically be fueled off or dedicated to various purposes, while having the option to fulfill the pinnacle loads, rapidly reallocating assets to the most demanding activities.

2.3.1 Cloud computing architecture

In distributed computing, assets are recovered from the web through electronic instruments and applications. This permits the clients to work remotely in light of the fact that the Cloud can be utilized as the "Web". In this way, it isn't handled as conventional redistributing. It is likewise called Massive Computing. In this, the assignment of the application must be dynamic. There is no compelling reason to introduce any sort of equipment and programming. The objective of distributed computing is to allow the clients to get to the information from all the advancements, applications with no deep information about them. In distributed computing engineering, the applications, information, and administrations all are put away in the Cloud by means of the web and run the applications and put away information by conveying the product assets as on-request benefits [7].

Cloud computing architecture can be divided into four layers, that is hardware layer, infrastructure layer, platform layer and application layer, as shown in Figure 2.1.

- **Hardware layer** : The physical assets of the Cloud are overseen by it. Controlling physical servers, switches, router, power framework is the duty of the equipment layer. The usage of the equipment layer is given in the server farm. This server farm contains a few servers that are interconnected through switches. A few issues happen in the equipment layer including adaptation to non-critical failure, equipment setup, traffic the board and assets of the executives [7].
- **Infrastructure layer** : It is a basic part of distributed computing. Framework layer depends on key highlights, for example, dynamic asset

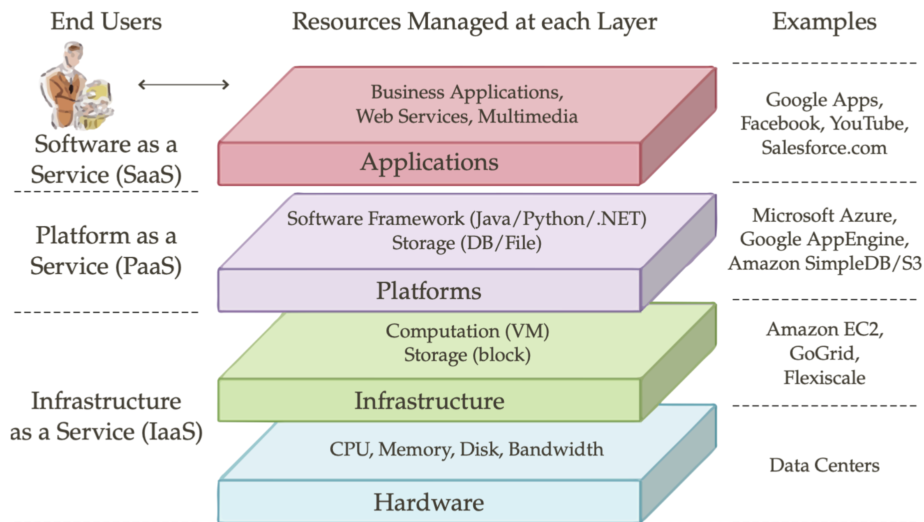

Figure 2.1: CLOUD COMPUTING ARCHITECTURE

tasks that are accessible through virtualization innovation. Framework layer makes the assortment of processing and capacity assets and partitions the physical assets by using virtualization procedures.

- **Platform layer**: It is based on the infrastructure layer. The fundamental idea of the stage layer is to limit the overhead of sending applications legitimately into VM compartments. For instance, Google AppEngine works at the stage layer to assign API bolsters for actualizing information stockpiling of the various web applications.

- **Application layer**: It is based on the top level of Cloud architecture. It is made out of real Cloud applications. Cloud applications have basic highlights to accomplish better execution, lower working cost, accessibility, and versatility. Therefore this engineering is more particular than other designs. Inexactly coupled ideas are utilized in each layer. This design grants distributed computing to convey a wide scope of utilization necessities while decreasing generally speaking overhead [7].

2.3.2 Deployment models

Computational assets accessible on the Cloud can be conveyed and disseminated to clients as per various models. The decision of one of those models, both

from the end-clients' and from the Cloud suppliers' point of view, is basically determined by the degree of dependability, adaptability, and security that is required.

- **Public Cloud**: It very well may be shared by different associations. Model Amazon, Google. Open Computing application stockpiling is made accessible to all associations. Assets are progressively appropriated over the web by means of web administrations.
- **Private Cloud**: This Cloud framework is committed to a particular association and can't be imparted to other associations. Private Cloud is progressively secure and increasingly costly as a contrast with an open Cloud and other blurring modes.
- **Hybrid Cloud**: It is blended of Public and Private Cloud and furthermore made out of in excess of two obfuscating modes. Association may have basic applications on open Cloud or private Cloud that are thoroughly relying upon requests. In a mixed Cloud, some portions of the applications administration framework are processed in private Cloud while the remaining part are processed in broad public Cloud. What's more, modes of Cloud computing are shown below [7].

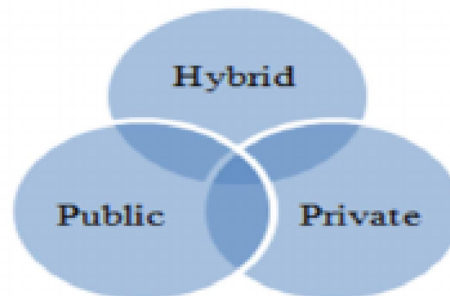

Figure 2.2: CLOUD COMPUTING MODELS

2.3.3 Cloud services

Cloud services can be divided in four main categories:

- **Infrastructure as a service (IaaS)**: It conveys a computer foundation that is a virtualized stage as help without purchasing programming and servers [6]. These infrastructures are for the most part given to clients by means of Virtual Machines, that permit them to send and run their own product. IaaS arrangements are generally utilized with regard to programming advancement and testing, other than for information reinforcement and capacity and in the structure of Big Data examination.
- **Platform as a service(PaaS)**: It permits application developers to have their administrations [6]. They can incorporate working framework support, databases, web servers, and programming improvement structures. They disentangle the procedure of programming improvement by maintaining a strategic distance from clients of the setup of the server framework.
- **Software as a service(SaaS)**: The application itself is given by a specialist organization. Programming can be utilized as help over the web without introducing programming on the client's computer [5]. This ensures the clients the likelihood to access totally different applications without the need to introduce or refresh them, streamlining upkeep and backing. Notable instances of this sort of administration are given by Google systems like Google Docs or Google Sheets, which permit overseeing archive and spreadsheets altering without the need to introduce physical applications.

All the administrations portrayed above are commonly proposed as on-request administrations and they are sorted out as indicated by a compensation-to-go, evaluating a model that represents the use of the administrations themselves. The associations' interest for Cloud administrations is developing exponentially, getting one of the best contributing needs.

As indicated by the IDC report, this positive pattern includes, as a matter of first importance, IaaS administrations, generally utilized in the Big Data investigation structures, with an expected Compound Annual Growth Rate of 32% by 2023.

2.4 Virtual Machine

Virtualization has become a significant tool in computer system design and virtual machines are utilized in various sub-disciplines extending from the

operating system to programming dialects to processor structures. By liberating engineers and clients from the conventional interface and resource limitations, VMs upgrade programming interoperability, framework invulnerability, and stage flexibility. Since VMs are the result of assorted gatherings with various objectives, be that as it may, there has been generally little unification of VM ideas. Thus, it is valuable to take a step back, consider the variety of virtual machine structures, and portray them in a brought together way, putting both the notion of virtualization and the types of VMs in context [15].

The idea of virtualization can be applied not exclusively to subsystems (for example, disks), yet to an entire machine. To implement a virtual machine, engineers add a product layer to a genuine machine to help the ideal design. Thus, a VM can bypass real machine similarity and equipment resource limitations [15].

2.4.1 VM allocation problems

The VM allocation problem is one of the central difficulties of utilizing the cloud computing worldview productively. Cloud computing envelops a few distinct setups and, relying upon this, the VM allocation issue additionally has various flavors [10].

Virtual machine allocation issues consist in deciding the best position of VMs on physical nodes, that can be arranged both on a private or public Cloud. In the accompanying, the entity who is accountable for taking care of the VM designation issue is meant as Cloud Provider. Most importantly, the interest for VMs that must be put onto physical machines can change after some time, just as the quantity of resources that those virtual machines require, as far as memory or computational power.

As an outcome, VMs can be relocated starting with one machine then onto the next furthermore, they can be turned on or off as per the interest. Both these forms are generally tedious, regardless of whether this time is frequently ignored in plans, particularly when it is extensively lower than the time required to run applications on the VMs themselves. This methodology will be misused likewise in this proposition work, where the execution times of Deep Picking up preparing jobs will be constantly thought to be certainly bigger at that point the movement times of Virtual Machines.

Physical machines have, thus, a fixed limit regarding assets they can have. Besides, the situation of various virtual machines on the equivalent physical host regularly prompts execution corruption, which ought to be considered when characterizing the ideal arrangement for the broken framework. The utilization of resources decides operational costs that are for the most part due to power utilization and are along these lines identified with the number of uses and the measure of time spent by various assignments on the VMs.

2.5 Heuristic approaches

The heuristic methodology is designed so as to construct great or close optimal solutions for complex optimization problems in a sensible measure of time. The expansive class of advancement issues can be partitioned into a progression of sub-classes, as per the chance of creating productive calculations to decide an answer.

2.5.1 Greedy algorithm

Greedy algorithms have been produced for a large number of issues in combinatorial optimization. For a large number of these greedy algorithms, exquisite most pessimistic scenario examination results have been gotten. These Greedy algorithms are regularly exceptionally simple to depict and code and they have extremely quick running times. On the other hand, the exactness of avaricious calculations is frequently unacceptable. When occasion size is large, precise arrangement approaches are commonly not pragmatic. In any case, a wide variety of metaheuristics have been effectively applied to take care of combinatorial optimization issues to within 5% or better of optimality [1].

The point of greedy algorithms is to discover sensibly great answers for the improvement issues they are tending to by methods for a sequence of locally optimal decisions. At any progression, the most encouraging move is chosen among the set of potential decisions, while never questioning past decisions.

Greedy algorithms are one of the well-known solutions for some advancement issues. As a simple algorithmic paradigm, it makes locally optimal decisions at each progression with the desire for accomplishing a globally good solution in sensible time [2].

Greedy algorithms work on problems for which it is true that, at every step, there is a choice that is optimal for the problem up to that step, and after the last step, the algorithm produces the optimal solution of the complete problem. Greedy algorithms identify an optimal substructure or subproblem in the problem. Then, determine what the solution will include (for example, the largest sum, the shortest path, etc.). Create some sort of iterative way to go through all of the subproblems and build a solution [13].

```
1:  S* ← ∅
2:  m ← f(S*)
3:  for i = 1, ..., N_max do
4:      S^i = set of available components
5:      s* ← new best component
6:      for s^i_j ∈ S^i do
7:          if f(S* ∪ {s^i_j}) < m then
8:              s* ← s^i_j
9:              m ← f(S* ∪ {s*})
10:         end if
11:     end for
12:     S* ← S* ∪ {s*}
13: end for
```

Figure 2.3: GENERAL GREEDY ALGORITHM

2.5.2 Randomized greedy algorithms

The fundamental issue identified with pure greedy methodologies is the fact that there is no assurance of getting an optimal solution. Indeed, the method of playing out an optimal decision in every single step does not really deliver

an answer that is optimal for the full issue. Regardless of whether this is reached, its optimality may hold just locally, since the procedure of never questioning the decisions characterized at past steps effectively leads to being stuck in local optima.

Using randomization can help to solve this problem by attempting to reach the global equilibrium of the problem. Randomness can be implemented by associating a probability distribution with candidate elements in S^i. In general, the level of randomness to be considered may be specified by weighing the likelihood of extracting an element ${s^i}_j$ from S^i based on the value it would add to the objective function.

A pseudo-code is stated in Figure 2.3 for the randomised greedy algorithm. As with the pure greedy version, each candidate variable is evaluated in set S^i. The probability ${p^i}_j$ of selecting ${s^i}_j$ as a new component s* is determined in a pure random approach only on the basis of a function of probability densities.

In addition, the objective feature is assessed in a mixed approach at the latest based on the new value, the candidate variable and ${p^i}_j$ are weighted.

$$v_j^i = f(S^* \cup s_j^i)$$

The new element s* is then extracted from S^i and added to the solution for the candidate.

2.5.3 Local search

Local search is a Metaheuristic strategy that is applied with the broadly useful of deciding a, conceivably optimal, solution for an optimization problem.

In many optimization problems, the state space is the space of all possible complete solutions. The local search consists in moving from a solution to another one in its neighbourhood according to some well-defined rules. For definiteness, we consider the problem of minimizing a function f(p) on a finite set of points p. This can be considered a general statement of a combinatorial optimization problem.

A local search strategy starts from an arbitrary solution $p \in \mathcal{P}$ and at each step n, a new solution p_{n+1} is chosen in the neighbourhood $V(p_n)$ of

the current solution p_n. This presupposes the definition of a neighbourhood structure on P; to each $p \in \mathcal{P}$ is associated V(p): a subset of P called the neighbourhood of p. For instance, if P is a set of binary vectors and $p \in \mathcal{P}$, a neighbourhood V(p) of p can be defined as the set of all solutions $p \in \mathcal{P}$ obtained from p by flipping a single coordinate from 0 to 1 or conversely [12].

Specifically, it is tended to as a local search in light of the fact that any conceivable move performed to show signs of improvement result is assessed distinctly as for a neighborhood set of solutions.

In the general system, there is no assurance for global optimality of the solution built by neighborhood search; be that as it may, this strategy is regularly applied to improve the aftereffects of other heuristic algorithms, for example, ravenous or randomized greedy methods.

Chapter 3

Methods

3.1 Scalable methods for the joint capacity allocation and scheduling of DL jobs

3.1.1 Centralized Model

The proposed Mixed Integer Linear Programming (MILP) is an extension of the Monolithic Model (MM) [6]. Specifically, it focuses on estimating the cost of first-end job delivery. This method is more appropriate for modeling the actions of an online scheduling problem. In reality, the framework is reconfigured not just every time a new job is submitted yet additionally every time a job's execution is finished. Since, from an online viewpoint, the only foreseeable event is the completion of a job and as all resources that are reallocated after a rescheduling, it is fair to bond the total cost to the first-end job's delivery expense.

Parameter	Description
J	The list of submitted job
N	The set of nodes
d_j	The dead line
v	Type of Virual Machine (VM)
g	Number of GPUs
j_n	Deployment cost
T_j	Tardiness of job

Table 3.1: PROBLEM PARAMETERS

3.1.2 Hierarchical approach

In order to improve the scalability of the problem, namely the scheduling of Deep Learning Training jobs on GPU-based Cloud Virtual Machines and the Centralized Model, a Hierarchical approach is proposed as follows, aimed at separating and resolving the analyzed problem via a series of local controllers.

The method is shown in Table 3.1 below. According to a Round Robin (RR) scheme, the number of reported jobs, denoted as normal by J, is split into a set of local queues, governed by a corresponding group of central controllers, denoted by the index $k \in \mathcal{K}$.

The RR algorithm assigns jobs in circular order to the local queues, so the union of all J^k, for $k \in \mathcal{K}$, forms the initial queue J. The same policy applies to the nodes in set N, which are divided into a series of subsets N^k, for $k \in \mathcal{K}$, each of which is managed by one of the local controllers.

Each controller K solves a joint resource allocation and schedule of jobs problem taking into account the J^k queue and the N^k subset, which can be equipped with different VM types and different GPU numbers.

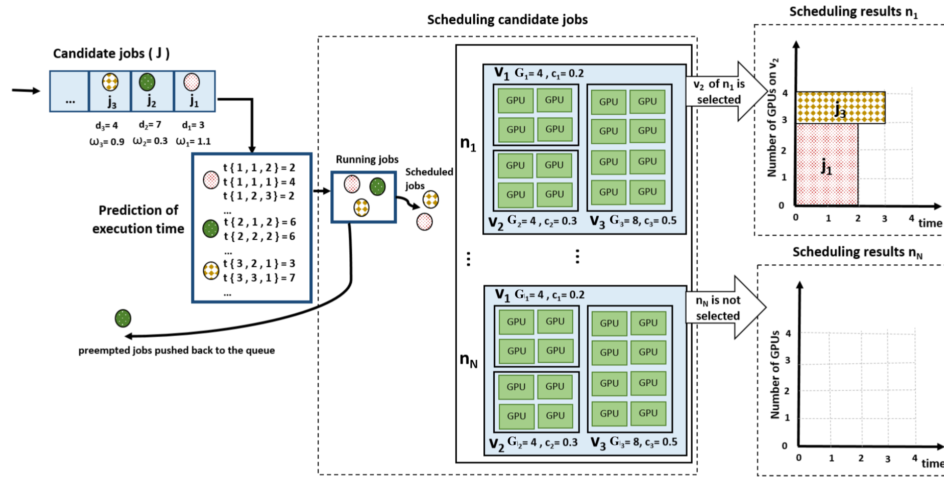

Figure 3.1: REFERENCE FRAMEWORK

3.1.3 Greedy algorithm

A Greedy algorithm is designed to solve the MILP problem. It expects to additionally improve, as for the result of the hierarchical approach, the scalability of the model, so that it can be employed to solve joint capacity allocation issues and job scheduling problems not just from the perspective of the Cloud end-clients, but also of the Cloud providers.

Greedy algorithms expect to discover sensibly optimal solutions for the optimization issue they are tending to by implies a sequence of locally optimal decisions. At each progression, the most encouraging local choice is selected, while failing to question past choices. So as to design a greedy algorithm for the problem, to indicate the rules used to decide the best local solution at each step, the following assumptions are considered:

- **First assumption**: jobs should be scheduled according to their pressure. For each job $j \in \mathcal{J}$, the pressure j is defined by determining how close it is to its deadline d_j when it is executed with the fastest configuration. Having denoted by T_c the current time when the scheduling is performed, the pressure of each job j is therefore computed as:

$$\Delta_j = T_c + \min_{v,g} (t_{jvg}) - d_j. \tag{3.1}$$

- **Second assumption**: the optimal configuration $(v, g) \in \mathcal{V} \times \mathrm{G}_v$ for each selected job is the cheapest configuration such that the job is executed before its deadline, if such a configuration exists, and the fastest available configuration if, independently from the selected setup, it is not possible to execute the job before its deadline.

- **Third assumption**: deployment costs increase linearly in the number of GPUs, as demonstrated by Cloud providers pricing models and the speedup of jobs' execution is sublinear in the number of GPUs, as observed in GPU-based application benchmarks.

The Greedy algorithm consists of three main phases: preprocessing phase, scheduling phase, post-processing phase.

1. **Preprocessing phase**
   First of all, jobs are sent, sorted by their pressure, and the current

STEP# 0 - PREPROCESSING

1: $T_c \leftarrow T_c + \bar{t}$
2: $\mathcal{N}_O \leftarrow \emptyset$ ▷ $\mathcal{N}_O$: set of open nodes
3: **for all** $j \in J$ **do**
4: **if** j has partially been executed in the previous time period **then**
5: **for all** $v \in \mathcal{V},\ g \in \mathcal{G}_v$ **do**
6: $t_{jvg} \leftarrow t_{jvg} \cdot (100 - \bar{c}_{pj})/100$ ▷ $\bar{c}_{pj}$: completion percentage
7: **end for**
8: **end if**
9: $\Delta_j \leftarrow \min_{v,g}(t_{jvg}) + T_c - d_j$
10: **end for**
11: $\mathcal{J}_s \leftarrow$ SORT_JOBS_LIST$(\mathcal{J},\ \Delta)$ ▷ Δ : pressures of all jobs

STEP# 1 - SCHEDULING

12: **for all** $j \in \mathcal{J}_s$ **do** ▷ $\mathcal{J}_s$: sorted queue
13: $D_j^* = \{(v, g) \text{ s.t. } t_{jvg} + T_c < d_j\}$
14: $(v^*, g^*) \leftarrow$ SELECT_BEST_CONFIGURATION(j, D_j^*)
15: assigned $\leftarrow$ ASSIGN_TO_EXISTING_NODE$(j, (v^*, g^*), \mathcal{N}_O)$
16: **if not** assigned **then**
17: **if** $|\mathcal{N}_O| < N$ **then**
18: $\mathcal{N}_O \leftarrow \mathcal{N}_O \cup \{\nu'\}$, where ν' has VM type v^* and G_{v^*} GPUs
19: Assign j to ν' with configuration (v^*, g^*)
20: **else**
21: ASSIGN_TO_SUBOPTIMAL$(j, \mathcal{N}_O)$
22: **end if**
23: **end if**
24: $\mathcal{J}_s \leftarrow \mathcal{J}_s \setminus \{j\}$
25: **end for**

STEP# 2 - POSTPROCESSING

26: **for all** $\nu \in \mathcal{N}_O$ **do**
27: **if** there exist idle GPUs on ν **then**
28: Look for an equivalent configuration (same VM type) with a lower number of GPUs
29: Assign the idle GPUs to the job with the highest speed-up
30: **end if**
31: **end for**

Figure 3.2: GREEDY ALGORITHM

time T_c is modified by adding the elapsed time t between the previous schedule and the new schedule, while the open node set N_O is initialized by the empty set. Then, consideration is given to the complete queue J of the submitted jobs. If a job has been partially executed in the preceding period, its execution time t_{jvg} must be modified for all possible values of $v \in \mathcal{V}$ and, therefore, $g \in \mathcal{G}_v$. Specifically, being c_{pj} the completion percentage of job j, its execution time is updated by replacing its value with:

$$t_{jvg} \cdot \frac{100 - \bar{c}_{pj}}{100}$$

After updating the execution time of all jobs submitted in J, their pressure is determined so that the list of jobs submitted can be sorted, by calling the corresponding function, generating a new queue denoted by J_s. In the sorted queue$J$ is placed before k if and only if $j > k$, i.e. job j deadline is more likely to be missed.

2. **Scheduling phase**
   In this step the best possible configuration is chosen for all queue jobs. For all jobs j in the sorted queue J_s, the set of configurations such that the job can be executed within its deadline is defined as:

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

   In the procedure, the set D_j* is used to select the best job j configuration, according to the following rules.

   - If D_j* is not empty , i.e., job j can be executed within the time limit, The best configuration is the cheapest one.
   - When, in effect, job j can not be executed within its time limit, regardless of the chosen configuration, the optimal setup for the system is the fastest.

   The assignment continues as follows, having denoted the appropriate configuration for job j by *(v*, g*)*.

   (a) Firstly, the algorithm tries to assign job j to an already available node With the right configuration *(v*, g*)*. It can be achieved in two different ways. First, a first-fit approach: let $\mathrm{N}_O \subseteq N$ be the set of already open nodes; if there is a node v $\in \mathcal{N}_\mathcal{O}$ whose VM and whose number of available GPUs is higher than or equal to g^*, then job j is allocated to v with optimal configuration. The second procedure is a best-fit approach instead of looking for the first open node with a compatible configuration and enough free resources, the best available node is selected through the procedure reported in Figure 3.3.

   Specifically, One of the main objectives is to minimize the amount of idle resources, thus open nodes whose type of VM is compatible with the one required by the current job are sorted by their saturation level. The saturation level has been described as the number of GPUs remaining idle after deploying job j on v with the

necessary optimum configuration *(v*, g*)* for each node v $\in \mathcal{N}_O$ equipped with a VM type $v*$ and such that it hosts sufficient free resources to allocate any GPUs to the job j. Under this rule, jobs are allocated to open nodes in order to saturate the node resources as much as possible, thereby reducing the total number of idle GPUs.

(b) If the assignment to an already open node is not feasible, but $|N_O| < N$, a new node ' is opened, with VM type v* and the maximum number G_v* of GPUs available. Job j is then allocated to v' with configuration (v*, g*) and the number of GPUs available on v' is changed as needed.

(c) Lastly, if *(v*, g*)* does not match in any open node and all available nodes are already open ($N_O \equiv N$), the job j is assigned to the sub-optimal configuration of the node v $\in \mathcal{N}_{\mathcal{O}}$. Two alternative assignment realizations to sub-optimal setups can be exploited. In the first version, which follows a best-fit approach, The job is assigned to the first configuration available on an open node. The second edition, which follows a best-fit approach, assigns the given job to the best of the sub-optimal configurations on available open nodes.

3. **Post Processing Phase**

The final step of the algorithm is a post-processing phase which aims at reducing the number of idle GPUs in open nodes, as much as possible. This objective is achieved through two distinct operations.

- First, the fact that two configurations are said to be equivalent if they have the same type of VM but different number of available GPUs, the algorithm tests whether it is possible to substitute the currently selected setup with an equivalent one on an open node, having a lower number of available GPUs.
- If the update presented in the previous step can not be performed or if there are still nodes with idle GPUs after it has been completed, the additional resources will be reallocated by assigning them to the job which gains the highest speed-up.

Greedy algorithm was implemented by means of a C++ code, to optimize efficiency in terms of execution time. As mentioned earlier, the ultimate purpose is to address problem instances in such a way as to consider a very large number of nodes and jobs.

Let J be the cardinality of the queue $\mathcal{J}$ of submitted jobs and let N be the cardinality of the set of nodes $\mathcal{N}$. Moreover, define $C = \sum_{v \in \mathcal{V}} G_v$ as the cardinality of the set $\mathcal{V} \times \mathcal{G}_v$. The overall complexity of lst:greedy can be computed as described in the following. First of all, the complexity of the preprocessing stage is given by two terms: the first, $\mathcal{O}(JC)$, is due to the update of all execution times, while the second, $\mathcal{O}(J \log J)$, is determined by the sorting operation. Overall, the complexity of preprocessing is therefore $\mathcal{O}(JC + J \log J)$.

As far as the scheduling stage is concerned, the first term influencing the complexity is due to the selection of the best configuration. If this operation needs to be performed only once, to determine the best configuration of a given job, because the assignment to a sub-optimal configuration follows instead the first-fit method there is need to build the set D_j^* explicitly. Therefore all the configurations available must be checked in this case and the resulting complexity is O(C).

If the assignment to a sub-optimal configuration is in effect the best choice approach in the best configuration selection may be performed many times during the algorithm. Therefore, at the outset, a more convenient option would be to construct specifically the set D_j. The selection of an ordered associative container specifies for this operation a complexity of *O(ClogC)* , but allows the actual complexity of the selection process to be reduced to *O (1)*.

For what concerns the assignment process, in both the first-fit and best-fit approach, the complexity is *O(N)*. The assignment to a new node, as long as the information about the last used node is always available, is reduced to O(G_{v*}), as the VM of type $v*$ with the maximum number of GPUs must be selected. Since *Gv* is, for all v $\in \mathcal{V}$, considerably smaller than all the other dimensions of the problem, the complexity of the assignment to a new node can however be reduced to *O(1).*

Finally, the complexity of the assignment to a suboptimal configuration depends on the chosen strategy. For the first-fit, if, in the worst case, no open nodes can host the incoming job, the complexity is $O(N)$. Following the best-fit approach, the complexity of the worst case, where all possible configurations must be examined, is given by $O(CN)$.Since both the selection of the best configuration and the assignment process are carried out in the queue J for all jobs, the above measured terms must always be multiplied by the number of jobs J.

Finally, the post-processing step has $O(N + J)$ complexity, determined; first of all, by replacing configurations which leave a large amount of idle resources and, secondly, by assigning still idle GPUs to the job which gains the highest speed-up.

3.1.4 Randomized Greedy algorithm

As mentioned earlier, Pure Greedy algorithms usually provide sub-optimal solutions, which can sometimes be of low quality. The procedure can be modified by introducing randomness in different stages of the pre-processing and the selection phases to enhance the robustness of the method and to find lower-cost solutions.

1. **Preprocessing stage**
   The pre-processing randomization consists of re-molding the sorted queue J_s, where the probability of swapping the location of two jobs in the queue is weighted by their priority. In particular, since the penalty for breaching the deadline is more expensive for higher-priority jobs, the likelihood of changing their order in J_s is lower, as jobs are more likely to be executed if they are at the queue's first positions.

2. **Scheduling stage**

   The randomization strategy exploited in the scheduling step is based on the pattern described for the Greedy Randomized Adaptive Search construction phase. In particular, a Restricted Candidate List (RCL) of elements is defined instead of choosing the element that has the best outcome with respect to the selection procedures described above and the new component of the solution is extracted from RCL at random.

```
1: function SELECT_BEST_NODE((v*, g*), N*)
2:     N̄ ← ∅
3:     for ν ∈ N* do
4:         v = VM type deployed on ν
5:         n = number of available GPUs on ν
6:         if v == v* and n ≥ g* then
7:             for ν′ ∈ N̄ do
8:                 v′ = VM type deployed on ν′
9:                 n′ = number of available GPUs on ν′
10:                if v′ == v and (n − g) < (n′ − g) then
11:                    ν is inserted in N̄ before ν′
12:                end if
13:            end for
14:        end if
15:    end for
16:    return a node randomly selected from a suitable subset of N̄
17: end function
```

Figure 3.3: SELECTION OF BEST NODE(RANDOM GREEDY)

First of all, in the selection process, instead of determining the best configuration, a small subset of candidate configurations is selected from D_j and the optimal configuration (v^*, g^*), is randomly selected from this subset; with a probability that is inversely proportional to the configuration cost itself.

Finally, assigning a job to an already open node is randomized starting with the best-fit approach and modifying the function used to select the best node. In fact, instead of scanning all open nodes and selecting the one that leaves the smallest amount of idle resources, open nodes are sorted to form the set N according to their saturation level, a subset of candidates is extracted from N and the best node is randomly selected from this subset with a probability that is inversely proportional to the number of idle GPU.

In the randomized case too, the overall structure of the Greedy algorithm remains the same. Nonetheless, the complete process, including pre-processing, scheduling, and post-processing phases, is repeated multiple times to find the best randomization solution.

3.1.5 Local Search

A step of local search can be performed to enhance the results obtained by a randomized greedy algorithm. The strategy followed in this work , in particular, prescribes saving the best solutions obtained by the randomized greedy in a set S and applying a step of local search to all of them, in order to find the best solution. Since the worst situation for a job is to terminate its execution after the prescribed deadline, thereby creating a tardiness penalty, the collection of jobs in tardiness are the first candidates that our local search often takes into consideration. There are three types of neighbourhoods:

1. Neighborhoods involving nodes hosting the jobs with highest expected tardiness, To improve the current solution by modifying the configuration selected from the node on which they run.
2. Neighborhoods which seek to reduce the total cost of the schedule by considering running and postponing jobs and attempting to exchange their state hunting for better solutions.
3. Neighborhoods which include a list of jobs to select improve movement by swapping the configurations assigned to them, according to different criteria.

All of those techniques are further detailed by identifying the necessary information criteria used to pick jobs or nodes which should be updated in configuration. In addition, a best-improving strategy is adopted, while parameterizing the number of random solutions to be saved in S, and the number of jobs or nodes to be considered in building the neighborhoods, so that their values can be adjusted to achieve better results.

The neighborhoods explored were designed as follows: First of all, Jobs whose execution, with the configuration selected, would violate the deadlines are sorted in decreasing order relative to the projected lateness. It allows neighborhoods experimentation to begin from the jobs whose penalty cost is likely to be the greatest, with the goal of reducing the expense of choosing a new design for the solution. Denoted the sorted list with J_T, the local search algorithm proceeds by exploring the neighborhoods as listed below. The first k elements of J_T are contrasted with the elements of another goal list during all of the discovery process. Only the first k or all the elements of the goal list are evaluated, depending on the neighborhood considered.

1. Two sets of jobs are considered: the first k elements of the J_T list mentioned above, of jobs sorted by their anticipated tardiness, and a list of k running jobs which, with the current configuration, does not infringe the deadline, sorted by their execution costs, in decreasing order. The algorithm goes through the selection of all possible pairs from those sets of jobs running on different nodes, and assessing the cost of a new schedule by swapping their configurations. In particular, if the two jobs in the pair, e.g. j_1 and j_2 are executed on nodes n_1 and n_2 respectively, with configurations (v_1, g_1) and (v_2, g_2), the proposed move consists of deploying j_1 on n_2 , with the VM type and the number of GPUs (v_2, g_2) originally assigned to j_2 and vice versa. As a candidate solution, the step providing the lowest cost is saved.

2. The first k jobs in the J_T list, sorted by the planned delay of work, are compared to the full list of running jobs, considered in decreasing pressure order. As in the previous case, all possible pairs of jobs running on different nodes are chosen, and the objective function value is evaluated after that the corresponding configurations have been swapped. As a candidate solution, the step providing the lowest cost is saved.

3. Unlike in the previous examples, the third possible neighborhood does not find pairs of running jobs, but explores the set of submitted jobs. Specifically, the collection of deferred jobs, reported in rising pressure order, is compared with the list of operating jobs, in increasing pressure order. If a running job has less pressure than a postponed job, their status will be swapped and the objective function reassessed.

4. Given the J_T list of jobs whose estimated tardiness cost is greater than zero, consideration is given to the set of nodes where the first k of these jobs will be deployed. The fourth neighborhood is created by attempting to allocate a more efficient type of VMs to those nodes, while retaining the same number of GPUs. Many jobs will have a lower execution time when allocated to other types of VMs, depending on their characteristics, whereas the performance of other jobs will get worse. Therefore, if the total execution times of all jobs deployed on that node decreases, a VM is said to be "more efficient".

5. As with the preceding neighborhood, the fifth aims to reduce the total timing cost by changing the configuration of nodes hosting jobs with the utmost tardiness predicted. The configurations chosen to form the neighborhood in particular have the same type of VM but a double number of GPUs. These are thus assigned to running jobs without altering the relative mutual power.

6. A similar technique is also implemented in the sixth neighborhood, where only configurations are considered with half the number of GPUs. To be able to share GPUs among running jobs while retaining the original mutual power, only nodes are considered so that all jobs are deployed on an even number of GPUs.

7. Eventually, the same list of nodes hosting jobs with the highest delay is evaluated and a new candidate solution is generated by changing the number of GPUs allocated to each job, with the constraint that the total number of GPUs needed can not surpass the amount of resources originally available on the node.

The value of the objective function created for all candidate solutions by exploring these neighborhoods, and the value obtained from the randomized greedy construction are always compared, so that only improving movements are exploited. Finally, the best among the solutions obtained with each neighborhood is chosen as a new schedule.

The overall complexity of exploration of the neighborhoods has to be considered when implementing the local search procedure and added to the complexity of the randomized greedy construction process. Specifically, a maximum number of jobs is then derived from the J_T list to investigate the neighborhoods, so that the number of jobs explored in the current iteration turns out to be the minimum between k and the number of jobs in lateness. Having constructed J_T and at the same time a list of k running jobs, considered in decreasing order of cost of execution, the exploration is carried out.

With concerns to the first neighborhood, from the first k jobs in J_T and in the list of jobs sorted by execution costs, all possible pairs are generated, resulting in a complexity of $O(k_2)$. In addition, the first k elements of J_T are contrasted with the list of running jobs in decreasing pressure order in

the second neighbourhood. Since the list J of jobs is sorted according to pressures at the beginning of the construction process listed in the previous sections, the pairs of jobs examined in this context are generated by comparing J_T's first k jobs to the full J, resulting in O (k_J). This is worth noting that k is selected as being considerably smaller than J. The list J of submitted jobs is considered also in the third district, where delayed and running jobs are compared, resulting in O (J_2) complexity.

In the fourth, fifth, sixth and seventh neighborhoods, the J_T list is used to drive node exploration, so those with the highest tardiness are considered to host the jobs.

In exploring the fourth neighborhood, the list of jobs running on the same node is analyzed for each of those jobs, if the corresponding node has not yet been visited. If $v \in \mathcal{V}$ is the type of VM deployed on the current node n, the maximum number of jobs running on n, under the worst-case scenario that all of them are allocated to a single GPU, is G_v, the total number of available GPUs on VM a type v. All possible configurations are evaluated for each job to decide whether there is a more efficient configuration that can substitute the current one.

The overall cost of this operation, being $C = v \in \mathcal{V}\mathcal{G}_v$ is given by $O(kG_vC)$. If such a configuration exists, all jobs running on the current node are scanned again to change their setup. Hence, under the assumption that all jobs are late deployed on different nodes, in order to carry out the analysis k times, the average exploration expense of the neighborhood is $O\ (kG_vC + kG_v)$ *and* $O\ (kG_vC)$.

In the case of the fifth and sixth districts a similar approach is used. However, because the types of VMs that can be deployed on the different nodes are the same for all jobs, the review of available configurations to find out whether there is a particular configuration with a double or half number of GPUs is done for each node only once. Therefore, the total cost for the fifth and sixth neighborhood is $O(kC + kG_v)$, under the hypothesis that all workers are deployed on separate nodes in late. Because $C = v \in \mathcal{V}\mathcal{G}_v$, by definition, is substantially greater than G_v for all $v \in \mathcal{V}$, this can be approximated to $O\ (kC)$.

Finally, in the seventh neighborhood, the running jobs are allocated a configuration with the same type of VM but a different number of GPUs. Consequently, because the available configurations are grouped by their VM from using an unordered associative container, the total exploration cost may be approximated to O (kG_v log C).

In both neighbourhoods, an incremental cost is defined by the objective function assessment required to decide if the new step represents an improvement on the cost of the original solution. The test can be carried out in two separate ways, as the suggested alteration might alter the identity of the first-end job on the network.

Indeed, the completion of execution of a job defines a rescheduling in an online scheduling process, thus the assessment of the current configuration is only true until this event occurs. An $O(J)$ expense is due to decide which is the system's first-end task. Having that information, if the first-end job is the same as in the original solution, the objective function value can be re-evaluated by measuring only the discrepancies between the original schedule and the proposed changes. Nevertheless, if the first-end job changes in the proposed new timetable, the objective feature must be recalculated for all workers, which specifies an additional expense of $O(J)$.

Chapter 4

Result and Simulation

4.1 Experiment Setting

The proposed models were developed in the special context of the Deep Learning training tasks to solve a joint capacity allocation and job scheduling problem.

In general, Deep Learning applications have a tremendous demand for resources, in terms of both computing power and storage. Consequently, when performed on GPU-based systems, they typically achieve considerable performance improvement, being deployed on various configurations according to their requirement, probably variable.

Specifically, the applications chosen to evaluate the proposed scheduling methods consist of heterogeneous Neural Networks (NNs) training tasks, the main applications we considered are as follows:

- **AlexNet**: is an extremely versatile model that can provide high accuracy on very complex datasets. Its architecture consists of five convolutional layers plus three layers which are completely connected. AlexNet is a deep Convolutional Neural Network (CNN) which is designed to recognize images [8].
- **Visual Geometry Group (VGG)**: VGG Neural Network has been designed to improve predictive accuracy by increasing the number of active layers as opposed to previous architectures. Its training process is highly, therefore the amount of computational power available on the system significantly affects its performance.
- **Deep Speech**: A speech recognition system based on the use of a Recursive Neural Network, the performance of which is determined primarily by the memory size and speed.
- **ResNet**: Has been designed to ease the deep network training process. The general purpose of a NN, H(x) as an underlying mapping to be fit by a few stacked layers, given an input x, ResNet modifies this method by estimating a residual function:

$$F(x) = H(x) - x$$

It is characterized by a balanced type of workload as compared to the former [4].

The three CNNs were chosen as targets for image classification tasks; the proposed experiments are planned because they are traditionally known as the best architecture in the ImageNet competition. In fact, they are commonly used in realistic contexts, owing to the implementation of the theory of paradigm learning. Since their architectures are heterogeneous, representative examples of the variety of architectures used for image classification and video processing in practice may be considered.

Several instances of the Deep Learning training applications listed, both implemented with PyTorch and TensorFlow frameworks, were considered, by varying the number and batch size of the epochs and thus with different characteristics in terms of the expected execution times. Such execution times were calculated by relying in particular on Machine Learning models based on linear regression; such models allow to learn the execution times of Deep Learning workers, beginning with a training set of experimental runs of the target applications themselves, with an average error of about 10% [5].

The joint capacity allocation and job scheduling issue considers different types of VMs as target execution platforms, whose characteristics are reported in Table 4.1. Six of the available VM types, namely NC6, NC12, NC24, NV6, NV12 and NV24 are based on Nvidia K80 and M60 GPUs, and are available in the catalog of Microsoft Azure. Three types of VM, using Quadro P600 or GTX 1080Ti GPUs, are based on a configuration of the in-house servers. Finally, the two VMs named "Standard NC24X" and "Standard NV24X" are not available as reference in the Azure catalog. However, they have been added, with time unit costs consistent with those of the elements selected from the catalog, in order to extend the spectrum of potential solutions.

4.2 Simulation

The general structure of the process used to solve VM capacity allocation and job scheduling problems is independent from the chosen approach and therefore is common to all the available methods. This consists mainly of three parts, namely a data generator, a simulator and a solver, as illustrated in Figure 4.1. In particular, the first is common to all the applied techniques,

VM type	GPUs type	Number of GPUs	Cost [$/h]
Standard NC6	K80	1	0.56
Standard NC12	K80	2	1.13
Standard NC24	K80	4	2.25
Standard NV6	M60	1	0.62
Standard NV12	M60	2	1.24
Standard NV24	M60	4	2.48
In-house server 1	Quadro P600	2	0.11
In-house server 2	GTX 1080Ti	8	1.13
In-house server 3	Quadro P600	8	0.44

Table 4.1: CHARACTERISTICS OF THE TARGET NODES

while the others have been adapted differently for the two cases of the Hierarchical approach and the heuristic methods.

The process of generating data has been implemented in Python. Given a list of inputs related to the characteristics of the required system, it allows all the data needed to run the simulation process described below to be generated.

Specifically, the parameters, that can be provided through a suitable configuration file, concern the number of nodes that will be available in the system, the total number of applications that be submitted along the simulation, denoted in the following as J_c, and the average inter-arrival time to be considered.

Starting from a pool of candidate programs, which features on all the available configurations in terms of execution times. A Python code selects J_c jobs at random to shape the $J_{candidate}$ package. In this step, as defined in section 4.1, all information relating to submission times, deadlines, and tardiness weights are determined. During the simulation process the applications in

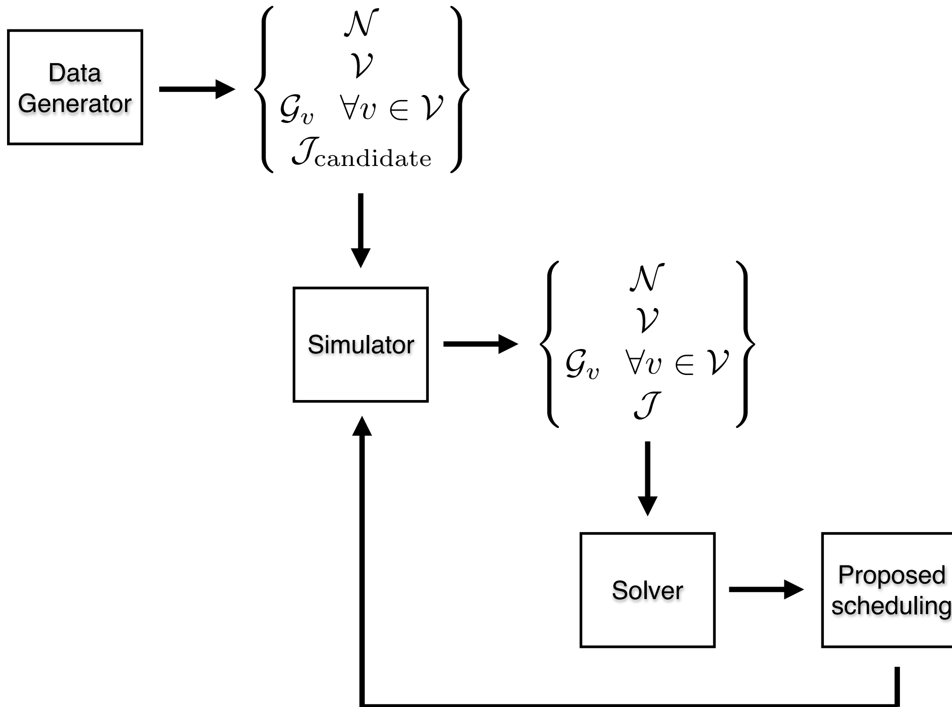

Figure 4.1: GLOBAL STRUCTURE OF THE SOLUTION PROCESS

$J_{candidate}$ will be selected to form the queue J of submitted jobs.

In addition, all the information related to the resources available in the system, namely the set N of nodes and the set V and G_v representing the catalog of available VMs and the GPUs that can be installed on each VM, are generated in this section and used to solve the instance provided below.

Simulation method is responsible for reproducing the actions of a real online program, which receives a series of requests and runs the current solver to decide the best schedule for all applications submitted. The simulator configuration is similar for the Hierarchical process and heuristic procedures, but it was introduced separately to better communicate with the underlying solver.

The simulation process consists mainly of a time loop, the iterations of which

are driven by the submission of new applications or the completion of their implementation. Specifically, the new jobs are extracted from $J_{candidate}$ based on their time of submission and inserted into the queue J. This queue is provided to the solver, as well as the information about available resources, which analyzes the current instance in order to determine the best scheduling. possible. This is then returned to the simulator, which measures the costs associated with the execution of the jobs and, if a task is completed, the potential fines due to violations of deadlines. The process ends when complete execution of all jobs.

All the results presented were obtained by running the implementations on a VM running on top of a server based on Intel Xeon E5-2640, using 32 cores and 32 GB of memory.

Many instances of the program that included a large number of nodes and submitted jobs were considered. The mean value equal to 30000s was selected having considered the minimum execution times of jobs, so that the scheduling process is guided, at least in the first part, by submissions of new jobs rather than by their finishing time. The mean time interval is divided by the number of available nodes, so that the workload of each node is constant as the network size increases.

Specifically, three sets of jobs were generated for each case, defined by a given size N and consequently by a fixed J, as mentioned earlier, to represent different submission scenarios. In addition, in the Randomized Greedy algorithms, the experiments were conducted with 10 different seeds for each generated job trace, in order to be able to discriminate the differences in the results determined by randomization from those by choosing another method.

The findings were compared to those of some of the key heuristic methods which can be used to solve problems with VM allocation and job scheduling. In particular, the algorithms used as benchmarks, called "first-principle methods", are:

- **First-In-First-Out (FIFO) method**: jobs submitted to queue J will be processed in their order of arrival as soon as resources are available. No preemption is permitted, so if a job is selected and assigned to a

certain configuration, that setup will be performed until it is completed.

- **Earliest Deadline First (EDF) method**: the queue J will be sorted with respect to the job deadline as soon as jobs are submitted, so that those who are more likely to violate it will be processed first. As far as the FIFO algorithm is concerned, it provides a better scheduling in terms of the tardiness costs, since this sorting decreases the frequency of violation of deadlines.

- **Priority Scheduling (PS) method**: the queue J is sorted with respect to the lateness weights of the jobs as soon as jobs are sent. This strategy aims at reducing the total tardiness expense of scheduling by handling in advance jobs that would assess a higher penalty if their deadline is broken.

  All of these algorithms are based on the premise that multiple jobs can not be shared between resources (nodes and VMs, other than GPUs). Furthermore, applications can not be preempted until allocated to a given configuration.

4.3 Comparison with first principle methods

To test the efficacy of the proposed strategies, the results obtained were compared, as in other literature proposals, to those measured using the so-called "first-principle methods", by considering a large collection of randomly generated instances; First-In-First-Out (FIFO), Earliest Deadline First (EDF) and Priority Scheduling (PS) methods are the three algorithms used as benchmarks for assessing the output obtained, both in terms of execution time and cost.

All these algorithms are based on the assumption that multiple jobs cannot share resources (nodes and VMs, other than GPUs). Furthermore, applications cannot be preempted once allocated to a given configuration. The outcomes of the analysis are summarized in the following pages, both for the Hierarchical Method and for the three heuristic algorithms.

4.3.1 Heuristic Approach

Both simulator and solver designed for the heuristic algorithms presented in Sections 3.1.3 and 3.1.4 have been implemented in a C++ program. Specifically, the Randomized Greedy method extends the solver by introducing an internal loop, that is in charge of performing the required number of iterations for the randomized construction. It consists of a mathematical optimization solver that can be applied to Linear, Quadratic and Mixed Integer Programming problems. The interface between the simulator and the solver has been implemented by relying on Pyomo [14], a Python-based modeling language that allows to define a symbolic model and to generate the instance that has to be solved.

In the case of the Hierarchical Model, the entire simulation process have been implemented as a Python library. In addition to the aforementioned procedures, it also involves the initial splitting of the set N of available nodes and the list of candidate jobs listed in Section 3.1.2.

This takes place at the beginning of the process, so that, if K is the number of local controllers, the list $J_{candidate}$ is split in K different portions before entering in the time loop representing the simulation.

This has been done in order to simulate a real system, such that all controllers run independently one from the others, by performing K different simulations, whose results in terms of computed costs are gathered only at the end of the whole process.

4.3.2 Hierarchical approach and Random Greedy Algorithm

The comparison between the Random Greedy algorithm presented in Section 3.1.4 and Hierarchical approach described in Section 3.1.2 highlights the great power of the former in terms of scalability. Have been tested by performing 1000 random iterations at each scheduling stage, with varying the mean value of inter-arrival times to (1000-40000).

A comparison of the results obtained with the Hierarchical approach and the Random Greedy algorithm , Figure 4.2 reports the total costs that obtains almost the same results.

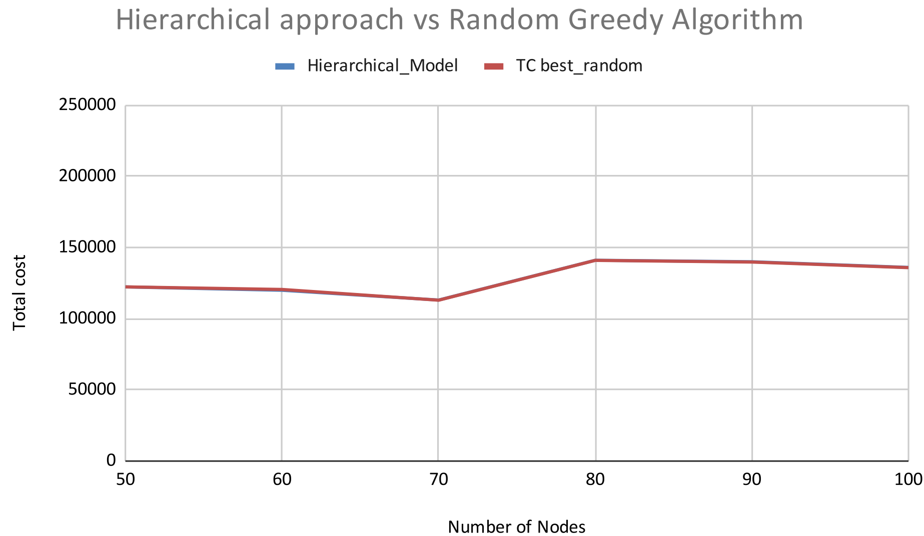

Figure 4.2: Comparison between Hierarchical Model and Random Greedy Algorithm

4.3.3 Pure Greedy and Randomized Greedy Algorithms

The Randomized Greedy algorithms, proposed in section 3.1.4 to enhance the results of the pure Greedy method, have been tested by performing 1000 random iterations at each scheduling stage, with varying the mean value of inter-arrival times to (1000-40000).

Specifically within the algorithm, the scheduling obtained through the proposed randomized construction is always compared, in terms of objective function value, with the solution of the Pure Greedy, so that the proposed solution is implemented only if its value, in the current scheduling step, has a promising outcome with respect to Pure Greedy, while solutions with a worse outcome are neglected.

Specifically, as shown in Figure 4.3, the percentage gain of Randomized Greedy algorithm with respect to EDF method is almost superimposable, with an average value of 95.99% compared to the 95.05%, of the pure Greedy algorithm.

It is worth to notice, as reported in Figure 4.3, that the average percentage gain of the Randomized Greedy method over the Greedy algorithm is of 9.56%.

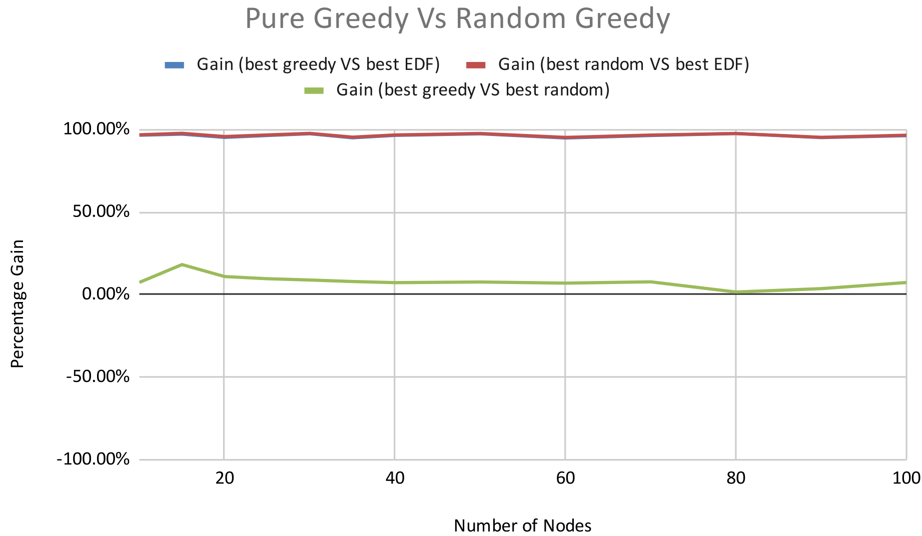

Figure 4.3: PERCENTAGE GAIN w.r.t EDF

4.3.4 Comparison between Centralized Greedy and Centralized Random Greedy Algorithms

The schedule developed by the Centralized Model at each stage have been implemented and worked on a specific framework to measure standards output with result obtained. The experiment results have been tested by performing 1000 random iterations at each scheduling stage, with fixed the mean value of inter-arrival times to 30000s.

The Figure 4.4 compares the real value obtained from the results of the complete simulation, measured at the conclusion of all the scheduling steps performed, with the average gain E_g. For each experiment, C_{rg} being the cost of the schedule determined by the Randomized Greedy algorithm and C_g being the cost of the schedule that would be returned by taking advantage of

the pure Greedy method, Randomized Greedy 's expected benefit in respect of Greedy is calculated as:

$$E_g = \frac{C_g - C_{rg}}{C_g} \cdot 100$$

The average gain of 3% over the Greedy algorithm. In particular, the

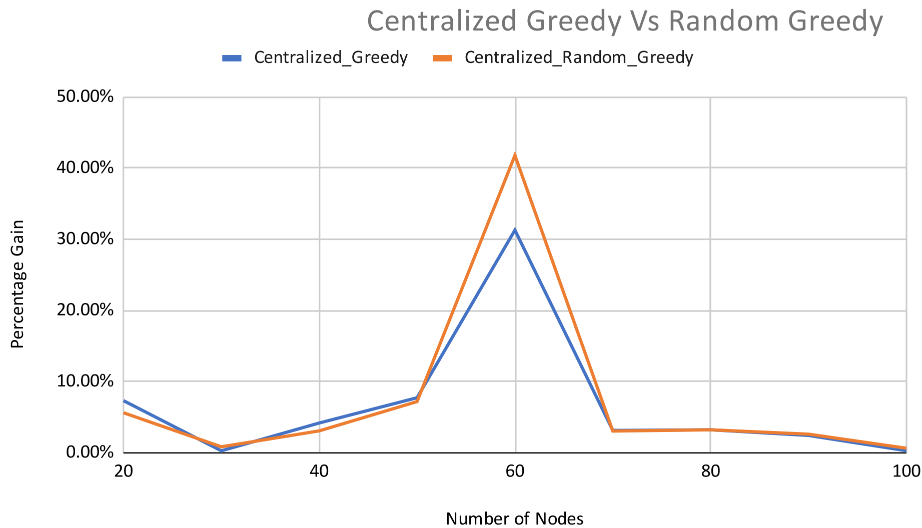

Figure 4.4: COMPARISON BETWEEN CENTRALIZED GREEDY AND CENTRALIZED RANDOM GREEDY

comparison between the total costs in all experiments with a number of nodes ranging from 55 to 75 shows that the Random Greedy improves the results.

4.3.5 Comparison between Centralized Greedy and Hierarchical Random Greedy Algorithms

The comparison between the Greedy algorithm presented in section 3.1.3 and the Hierarchical approach described in section 3.1.2 highlights the great power of the former in terms of scalability. The experiment results have been tested by performing 1000 random iterations at each scheduling stage, with fixed the mean value of inter-arrival times to 30000.

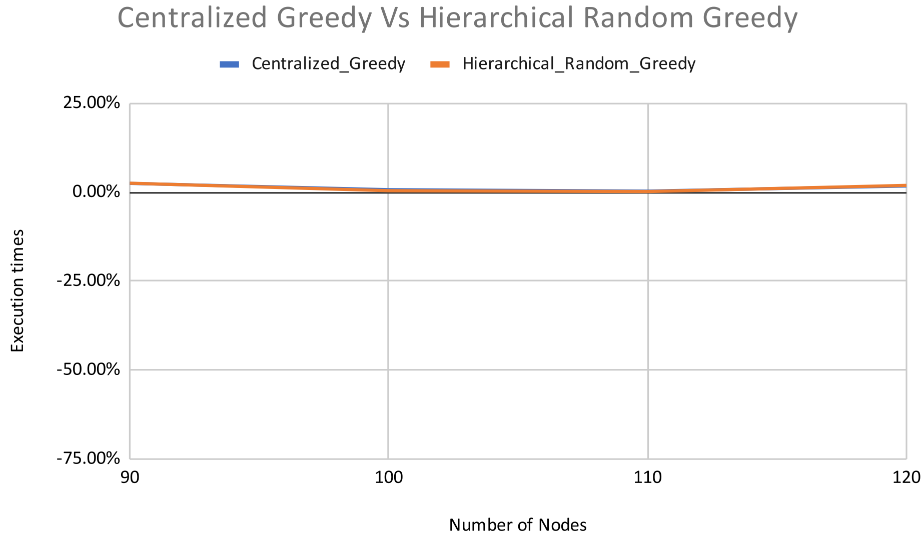

Figure 4.5: (a) COMPARISON BETWEEN CENTRALIZED GREEDY AND HIERARCHICAL Random Greedy Algorithms

Figure 4.5 reports the execution times of each scheduling step with the Hierarchical approach. Due to the splitting in local queues, the average execution times remain almost constant for the Hierarchical approach. it is relevant to notice that the scheduling process for each local controller is independent of what happens to the other queues. When a new job is submitted to the central queue J, it is assigned to one of the K local queues according to the Round Robin policy described in Section 3.1.2, so that the problem is solved only for the corresponding controller k.

A comparison of the results obtained with the centralized Greedy and the Hierarchical Random Greedy algorithms, both in terms of average gain and the cost of schedule, is shown in Figure 4.5 both Centralized Greedy and Hierarchical Random Greedy they are obtaining the same cost-related results.

The experiment results are performed as shown Figure 4.6 jobs J splitting in queues Q , The queues are created the first job j_1 is sent to the first queue q_1. the second job j_2 is sent to the next queue q_2, Then the next job j_3 is sent to the next queue q_3, Usually we will not have the same number of jobs and queues , so the job is assigned into the queue again until all jobs have

been completed. The pressure P defines jobs J, the pressure computes how much a job is close to the deadline, because the closest to the deadline d_j must be processed as first.

$$P = Minimun_execution_time - d_j$$

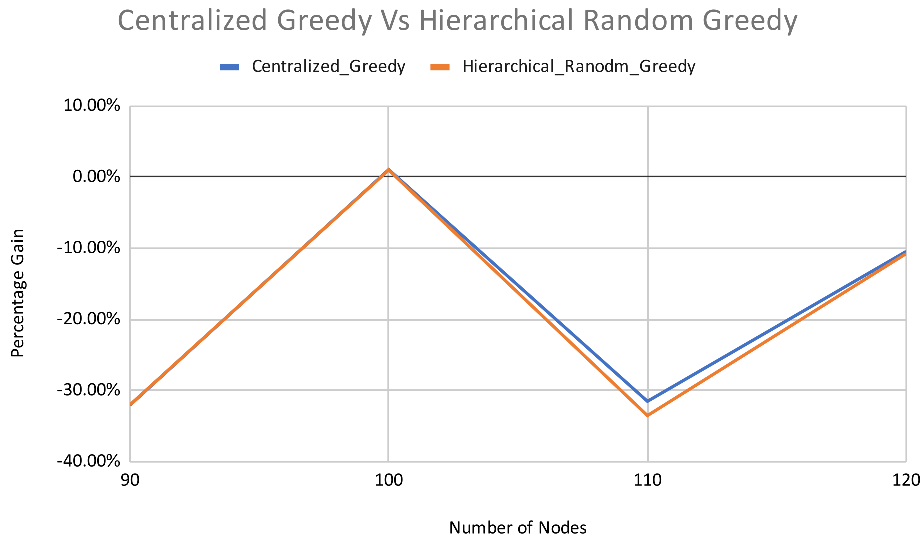

Figure 4.6: (b) COMPARISON BETWEEN CENTRALIZED GREEDY AND HIERARCHICAL

Specifically, as show Figure 4.6 The Hierarchical Random Greedy is faster than the Centralized Random Greedy, but it obtains the percentage gain -15% which is very poor results, so it's not worth using it.

4.4 Discussion

The Centralized Randomized Greedy method has been proven to be the best approach in all the considered scenarios for the problem analyzed from the perspective of cloud end-users. but it takes a lot more time than pure Greedy and Hierarchical Random greedy.

In both Greedy and Random Greedy obtaining the same cost-related results, it is also the execution time is very fast. The Hierarchical Random Greedy is

faster than the Centralized Random Greedy, but it obtains really bad results. Therefore it is not recommended to use it.

The results of the scalability analysis highlight the great power of all the proposed methods with respect to the initial work applied in [6], enabling to extend the experimental campaign to instances involving hundreds of nodes. Specifically, while the Monolithic Model required a computational time of nearly 14 minutes to solve the largest instance, including 40 nodes, the Hierarchical Model guarantees an average execution time of 40 seconds for all instances up to 100 nodes.

Even better results have been obtained with the Greedy, Randomized Greedy and Local Search methods, in 0.0038, 3.04 and 3.15 seconds, respectively. The speed-up obtained with all the proposed approaches over the initial Monolithic Model guarantees therefore a reduction of the required computational time by one or two orders of magnitude, while the Randomized Greedy algorithm have been proved to be more than 6 times faster than the Hierarchical Model, on average.

The analysis was extended to the case of Data Center environments. The comparisons performed in the simplified context of homogeneous Data Centers enabled to assess the promising behaviour of both the Hierarchical Model and the heuristic approaches.

Specifically, the pure Greedy method remains 40 times faster than the Hierarchical approach. The Randomized Greedy algorithm obtains, instead, promising results for instances considering up to 600 nodes, while a lower execution time can be achieved, in case of larger systems, by the Hierarchical Model.

Chapter 5

Conclusion and future work

The core objective of this thesis work has been the analysis of scheduling algorithms for deep learning training jobs running on virtualized GPU-based clusters.

The solution of this problem represent great challenges, exacerbated, in the envisioned scenario, by the fact that the problem is setup in an online setting, where multiple Deep Learning training jobs are submitted in a continuous fashion, so that no scheme can be detected in their arrival times or characteristics, particularly in terms of priority.

The main goal of Mixed Integer Linear Programming (MILP) models was to design an optimal scheduling for jobs, that would allow to minimize the overall execution costs, while meeting the constraints related to system capacity and applications deadlines. Resource sharing have bee designed in such a way that multiple tasks could be deployed on the same machine, with a variable number of dedicated GPUs.

The Hierarchical approach developed to solve the proposed model, discussed in section 3.1.2, aimed to tackle the solution of the problem in a hierarchical fashion,to reduce the dimensionality by splitting the Monholitic MILP formulation in a set of smaller sub-problems. In addition, three different heuristic methods, inspired to Greedy and Local Search techniques, have been developed, as reported in Sections 3.1.3 , 3.1.4 and 3.1.5, in order to further enhance the performance obtained with the Hierarchical approach, while maintaining the quality of the results in terms of scheduling costs.

The results, extensively discussed in Chapter 4 highlight the effectiveness of the proposed approaches, both in terms of scalability and quality of the identified solutions.

The review of the results obtained in all the considered scenarios by the different methods, as described in Section 4.4, allows one to conclude that the models and approaches built for this thesis work completely achieve the goal of providing flexible strategies for the solution of the problem of joint capacity allocation and job scheduling.

Future work will focus on further improving the approach scalability in order to manage clusters and disaggregated hardware resources of very large scale data centres.

Bibliography

[1] Carmine Cerrone, Raffaele Cerulli, and Bruce Golden. ?Carousel greedy: a generalized greedy algorithm with applications in optimization? In: *Computers & Operations Research* 85 (2017), pp. 97–112.

[2] Wanru Gao et al. ?Randomized greedy algorithms for covering problems? In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2018, pp. 309–315.

[3] Eugenio Gianniti, Li Zhang, and Danilo Ardagna. ?Performance prediction of gpu-based deep learning applications? In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2018, pp. 167–170.

[4] Awni Hannun et al. ?Deep speech: Scaling up end-to-end speech recognition? In: *arXiv preprint arXiv:1412.5567* (2014).

[5] Kaiming He et al. ?Deep residual learning for image recognition? In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[6] Arezoo Jahani et al. ?Optimizing on-demand GPUs in the Cloud for Deep Learning Applications Training? In: *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. IEEE. 2019, pp. 1–8.

[7] Shalini Joshi and Uma Kumari. ?Cloud computing: Architecture & Challenges? In: ().

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ?Imagenet classification with deep convolutional neural networks? In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. ?Deep learning? In: *nature* 521.7553 (2015), pp. 436–444.

[10] Zoltán Ádám Mann. ?Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms? In: *Acm Computing Surveys (CSUR)* 48.1 (2015), pp. 1–34.

[11] Arnab Paul and Suresh Venkatasubramanian. ?Why does deep learning work?-a perspective from group theory? In: *arXiv preprint arXiv:1412.6621* (2014).

[12] Marc Pirlot. ?General local search methods? In: *European journal of operational research* 92.3 (1996), pp. 493–511.

[13] Artem Potebnia. ?Representation of the greedy algorithms applicability for solving the combinatorial optimization problems based on the hypergraph mathematical structure? In: *2017 14th International Conference The Experience of Designing and Application of CAD Systems in Microelectronics (CADSM)*. IEEE. 2017, pp. 328–332.

[14] *Pyomo*. URL: http://www.pyomo.org (visited on 04/02/2020).

[15] James E Smith and Ravi Nair. ?The architecture of virtual machines? In: *Computer* 38.5 (2005), pp. 32–38.