POLITECNICO DI MILANO

Master's Degree in Nuclear Engineering

Benetti Alessandro

PATH RELINKING AS INTENSIFICATION STRATEGY FOR JOB SCHEDULING

Advanced Programming for Scientific Computing

Contents

1 Introduction **2**

2 Problem statement **2**
- 2.1 Greedy algorithm . . . 4
- 2.2 Greedy Randomized Adaptive Search Procedure (GRASP) . . . 5
- 2.3 Path relininking . . . 7

3 Code Description **8**
- 3.1 Elite Pool Construction . . . 9
- 3.2 Relinking . . . 9
- 3.3 Get moves . . . 10
- 3.4 Explore step . . . 11

4 Tests **14**

5 Results **14**
- 5.1 System loads and Data analysis . . . 14
- 5.2 General results . . . 17
- 5.3 Path relinking overhead . . . 18
- 5.4 Validation and other changes . . . 19

6 Conclusions **20**

References **21**

1 Introduction

Deep learning is a branch of artificial intelligence that has seen a rise in popularity in recent years. It can be used to solve complex problem like speech recognition and image classification, however the training of a deep learning model require enormous amount of computational power. This need can be partially satisfied by using GPUs as parallel processor, but the cost of buying and utilizing this type of hardware is still remarkably high. Moreover, different models require different GPU types for their optimal training: there isn't an optimal setup to train every single type of deep learning model. For this reason cloud computing represent an optimal solution for many industries, as they can rent the best type of hardware necessary for the training, instead of actually buying it. The problem analyzed takes into consideration a cloud provider, whose objective is to train multiple deep learning models by sharing its limited amount of resources among them. Previous works [1] dealt with creating the infrastructure to simulate this job scheduling problem, while this project deals with further optimizing the resource sharing part using an intensification strategy known as path relinking.

2 Problem statement

As stated in the introduction paragraph, this works takes into consideration a cloud provider that has to manage the training of multiple deep learning jobs using a server farm with a limited number of resources. The resources are represented by nodes: physical machines in the server farm that can activate their own Virtual Machine (VM) from a catalog listed in table 1. A virtual machine simulate a real operating system, characterized by a certain number and type of GPUs, where a job can actually be trained. A single VM can train multiple jobs, but a single job cannot be trained on multiple VMs simultaneously.

VM Type	GPU type	GPUs Num	Costs[$/h]
Standard NC6	K80	1	0.56
Standard NC12	K80	2	1.13
Standard NC24	K80	4	2.25
Standard NC24x	K80	8	4.48
Standard NV6	M60	1	0.62
Standard NV12	M60	2	1.24
Standard NV24	M60	4	2.48
Standard NV24x	M60	8	4.96

Table 1: Table of the possible VMs

The time required to complete the training of a job is a function of the hardware (Setup) chosen. Generally, this training time is different for each job, and assigning

multiple GPUs to a single job never cause its training time to reduce linearly, due to a communication overhead between the hardware components. This concept is presented in table 2, where some of the possible setups for the job J1 are listed, along with the predicted training time.

Job ID	VM type	GPU type	GPU used	VM GPUs	Execution Time[s]
J1	Standard NC6	K80	1	1	81990.5
J1	Standard NC12	K80	1	2	81990.5
J1	Standard NC24	K80	3	4	74829.8
J1	Standard NV6	M60	1	1	73837.9
J1	Standard NV24x	M60	7	8	62787.1
J1	Standard NV24x	M60	8	8	58879.1

Table 2: Execution times in different setups. A job can be trained using only a part of the GPUs that a VM has to offer

The jobs are also characterized by a deadline, a time limit imposed by the client that if surpassed will cost the could provider a quantity proportional to the extra time required to complete the job. This deadline in the simulation is chosen uniformly between [t_{min}, 3*t_{min}], where t_{min} is the fastest possible training time of the job, allowing to always complete a job before its deadline, if enough resources are available. Each time a job is submitted to the server, or a job is completed, the cloud provider perform the scheduling of resources. This process, as well as the general structure of the data server, is reported in fig 1. Since the number of possible schedules is enormous, especially for large systems, using heuristic techniques is necessary to generate a near optimal solutions in a reasonable amount of time.

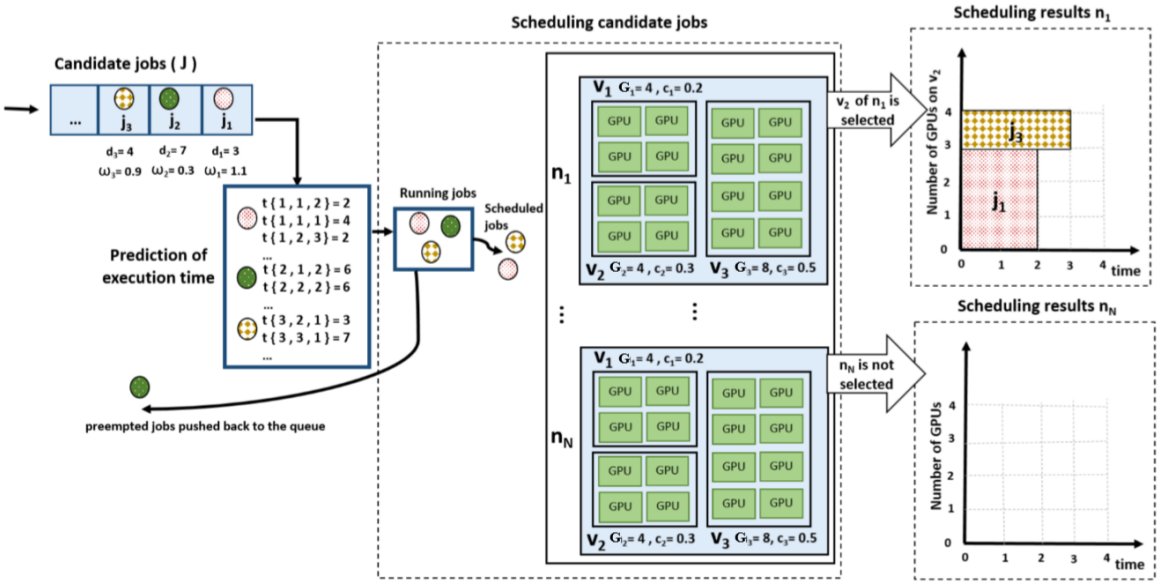

Figure 1: Reference framework. In the example, J = j1,j2,j3, with deadlines equal to (3,7,4), respectively. The values of execution times tjvg are chosen by way of example. Nodes n1 to nN can be equipped with three types of VMs: v1, with 4 GPUs and cost c1 = 0.2$/h, v2, with 4 GPUs and c2 = 0.3$/h, and v3, with 8 GPUs and c3 = 0.5$/h. Jobs j1 and j3 are deployed on n1 with VM v2. j1 runs on 3 GPUs, while j3 on 1 GPU. Job j2 is sent back to the queue and no other nodes are selected. Courtesy of [1]

2.1 Greedy algorithm

Among the various heuristic techniques that can be employed to solve a job scheduling problem, the Greedy approach is a proven method that produce good results, while being computationally inexpensive. The aim of these type of algorithms is to find near optimum solutions by following a series of local optimal choices. At any point this algorithm perform the most promising choice among a set of all the possible decisions, while never questioning the previous ones. Let any solution S of an optimization problem P be characterized by a set of n elementary choices, so that is possible to write $S = \{s^1, ..., s^n\}$, where each component s^i belongs to a family $\boldsymbol{S}$. Let P be defined as a minimization problem, suppose for example that the optimal solution of P is:

$$S^* = \arg\min_S f(S) \tag{2.1}$$

Greedy algorithm then try to find the optimal solution S* trough the following iterative procedure reported in algorithm 1:

Algorithm 1: Greedy algorithm general procedure for a minimization problem

$\widetilde{S} \leftarrow \emptyset$
m $\leftarrow f(\widetilde{S})$
for i = *1, ..., n* **do**
  S^i = set of available components
  s* = new best component;
  for *all* $s^i \in S^i$ **do**
    if $f\left(\widetilde{S} \cup \{s^i\}\right) < m$ **then**
      s* $\leftarrow s^i$
      m $\leftarrow f\left(\widetilde{S} \cup \{s^i\}\right)$
    end
  end
  $\widetilde{S} \leftarrow \widetilde{S} \cup s^*$
end

At each step i the algorithm considers a set $\mathbf{S}^i$ of the available components. The selected element s* to be added to the current solution $\widetilde{S}$ correspond to the local optimum in the current iteration, satisfying the condition:

$$s^* = \underset{s^i \in S^i}{\arg\min} f\left(\widetilde{S} \cup \{s^i_j\}\right) \tag{2.2}$$

In our job scheduling problem, the solution $\widetilde{S}$ is then the schedule, composed of the various setup s* of the jobs. After the job queue has been ordered in terms of pressure, the greedy algorithm assign the resources in order, by selecting the setup that is the cheapest available for the job at each step. Since performing a greedy choice every time we need to assign the resources does not guarantee reaching the global optimal solution, the next logical step is to insert some randomness in the setup selection process. This will allow to to make sub-optimal choices for some jobs in the queue, hoping to achieve an overall better solution later on in the scheduling.

2.2 Greedy Randomized Adaptive Search Procedure (GRASP)

GRASP is a multi-start, meta-heuristic algorithm used for combinatorial optimization problems [2]. It basically consists of two phases: construction and local search. GRASP is an iterative procedure, whose algorithm is reported in 2. Instead of always selecting the best element s$^i \in$ S^i, it blends greedy and random constructions by using greediness to build a restricted candidate list (RCL), and randomness to select an element from it. The

RCL is constructed by first calculating all the incremental costs c(s^i) for each element s^i $\in S^i$. This incremental cost can be viewed as the impact on the objective function f(S) of the previous section. Then, the RCL select the best elements from S^i, creating a set $\{s^i \in S^i : c^i_{min} < c(s^i) < +\alpha^*(c^i_{max}\text{-}c^i_{min})\}$. Other approaches for this RCL construction are possible, in the code for example, the RCL is cardinality-based, and is made up of the k top-ranked elements. After this list is constructed, an element s^* is randomly selected from it and added to the solution $\widetilde{S}$.

Algorithm 2: GRASP general procedure for a minimization problem

```
S ← ∅
Q ← g(S)
for j = 1, ..., k do
    S̃ ← ∅
    m̃ ← f(S̃)
    for i = 1, ..., n do
        S^i = set of available components
        s* = new best component;
        for all s^i ∈ S^i do
            evaluate the incremental cost c(s^i)
            c^i_min ← min(c^i_min, c(s^i))
            c^i_max ← max(c^i_max, c(s^i))
        end
        RCL ← ∅
        for s^i ∈ S^i do
            if c(s^i) ≤ c^i_min + α(c^i_max - c^i_min) then
                RCL ← RCL ∪ {s^i}
            end
        end
        pick s* at random from RCL
        S̃ ← S̃ ∪ {s*}
    end
    LocalSearch(S̃)
    if Q > g(S̃) then
        Q ← g(S̃)
        S ← S̃
    end
end
```

The parameter alpha dictates how greedy are the generated solution from GRASP, since for $\alpha = 0$ correspond a full greedy algorithm, while for $\alpha = 1$ a fully randomized one. A great advantage of GRASP is that it is easily tuned, as its only two parameters are α and the number of iterations performed. After each solution $\widetilde{S}$ is generated, a step of

local search deals with further optimizing the results. This step in the code however is substituted by a post-processing step that deals with leaving no idle GPU is the VMs. Many possible solutions $\widetilde{S}$ are generated in the GRASP procedure, and the one that minimizes the proxy function $g(S)$ the most is deemed to be the best one. In previous works this proxy function was based on the minimization of the cost in the current time step. This logic however did not guarantee reaching the global optimal solution at the end of the simulation, since being conservative on resources at come point, could cause the next time steps to be extremely expensive. For this reason, in this project a new proxy function was created 2.3:

$$g(S) = \sum_{j=1}^{n} \frac{K_j}{c_j + t_j} \tag{2.3}$$

This function tries to maximize the completion of each job with respect to its costs. K represent the maximum execution time of the job, a scaling factor introduced to give the same importance to every job in the queue. The denominator is then composed of two contribution, the first (c) is related to the cost of the virtual machine the job will be trained at, while the second (t) is equal to the costs due to any type of deadline violation we could incur using the selected setup. This function then tries to select the schedule that uses the resources in the most cost efficient manner, completing as much jobs as possible using the cheapest setup available. The units of measure are in [s/$], so we can still use it for the GRASP, but it will make the problem a maximization one. While GRASP correctly bring the global costs of the schedule down, its results can be further improved using intensification strategies.

2.3 Path relininking

Path-relinking is an intensification strategy that can be used to explore the trajectories connecting two elite solutions obtained by GRASP [2]. Starting from one elite, it explores the path leading towards the other, in the search for better solutions. The path relinking process for a maximization problem can be summarized in the pseudo code reported below.

Algorithm 3: General Path Relinking procedure for a maximization problem

Data: Starting solution S_s and target solution S_t
Result: Best solution S^* in path from S_s to S_t
$D \leftarrow$ symmetric difference $\Delta(S_s, S_t)$
$g^* \leftarrow \max\{g(S_s), g(S_t)\}$
$S^* \leftarrow \text{argmax}\{g(S_s), g(S_t)\}$
$S \leftarrow S_s$
while $D \neq \emptyset$ **do**
  $m^* \leftarrow \text{armgmax}\{g(S \oplus m) : m \in D\}$
  $D =$ symmetric difference $\Delta(S \oplus m^*, S_t)$
  $S \leftarrow S \oplus m^*$
  if $g(S) > g^*$ **then**
    $S^* \leftarrow S$
    $g^* \leftarrow g(S)$
  end
end

It first starts by calculating the moves necessary to reach the guiding solution from the starting solution, that represent the possible paths. Then, if the pool D of moves is not empty, it applies the move m^* that maximise the proxy function $g(S \oplus m)$ to the solution S. A move is always applied, even if the quality of the proxy is lower than the one in the previous step, allowing path relinking to explore even uphill scenarios, that may lead to a global optimum later on. The modified solution S then try to update the best solution found in the relinking, by comparing the proxy function g(S) with the best one g^*. This general framework can then be adapted to different scenarios, that may sacrifice some effectiveness for the sake of efficiency. First Resende [3] reports that since the neighborhood of the initial solution is much more carefully explored than the guiding one, starting the algorithm from the best solution available in the pool usually bring better results in less time. Additionally, for the same reason, most of the improvements are found at the beginning of the relinking, meaning that we can actually explore only the first part of the path without losing too much on effectiveness. Of course, applying these two choices will limit the quality of the relinking, but given then online constraints of this problem, it is a reasonable compromise. The path relinking developed in this project will then be used as a post optimization mechanism to the pool of elite solutions generated by the GRASP. It should be noted however that the code discussed below is made to be flexible, so employing the different styles of relinking reported in [2] is only a matter of changing some parameters or writing few lines of code.

3 Code Description

As stated in the previous paragraph, path relinking is composed mainly of two phases: Move selection and exploration. These two functions, along with the general relinking

procedure will be described in this section. Most of the code developed for this project is in the class "Path_relinking", and the additional python scripts for the data analysis and results validation are in the "Notebook" folder.

3.1 Elite Pool Construction

The first step to perform path relinking is actually generating a pool of elite solution. Since "Path_relinking" is a derived class of "Random_greedy", we can utilize the function of the latter for this step. This procedure is almost the same reported in 2, with the only difference that due to the nature of the proxy function 2.1, this is a maximization problem, instead of a minimization one. To ensure that the pool of elite is sparse enough, the parameter alpha controlling the RCL has been increased from 0.05 to 0.3, allowing the path relinking to explore more freely the solution space. The size of the elite pool is arbitrary, but for this project, 10 was found to be a good compromise between speed and performance. After this elite pool has been generated, path relinking can now begin between the best elite and the other solutions.

3.2 Relinking

The core of the path relinking is performed in the relinking_step function. It essentially follows the algorithm shown in 3, but apply some slight modifications to improve the speed of the algorithm. Due to the nature of the problem, this is a recursive function, that takes as an input a starting and a guiding solution, and will modify the first until reaching the latter, or until the maximum number of steps have been taken. Each step, the algorithm calculates the list of possible moves available between the two schedules using the function "Get_moves", and explore the effects of applying each one in terms of proxy function difference in "Explore_step". The move that yields the best result is then applied to the starting schedule, and the others are added to an unordered map called "explored moves". This map has the only purpose of avoiding exploring the same moves in different steps of the relinking, speeding up the whole process. The value of the proxy function of the modified schedule is then compared to the best one, to check if the best solution found in the relinking has to be updated or not. At the end of this whole process, the best schedule is then used to update the pool of elite solutions.

```
void
Path_relinking::relinking_phase(job_schedule_t &source,
                                const job_schedule_t &target,
                                double relinking_best,
                                double improvement,
                                unsigned DEPTH)
{
  std::pair<double, std::pair<Job, int>> step_best = {};
  pool moves = get_moves(source, target);
  double fitness=0;

  //If a move is possible, perform the linking
  if((!moves.empty()) && DEPTH>0){
```

```
14      //Consume a relinking move
15      DEPTH--;
16      //Explore the moves (not permanently)
17      for (auto move : moves)
18      {
19        //Check if the move has already been explored
20        auto fit = explored_moves.find(move);
21        if(fit==explored_moves.end()){
22          fitness = explore_step(source, target, move, FUTURE_SIGHT);
23          explored_moves[move] = fitness;
24        }else{
25          fitness = fit->second;
26        }
27        //Find the best step fitness & the associated move!
28        if (fitness >= step_best.first || step_best.first==0){
29          step_best = {fitness, move};
30        }
31      }
32      //Apply the best move (Permanently)
33      improvement += explore_step(source, target, step_best.second, false
        , true);
34
35      //Update the best solution found in the linking if possible
36      if (improvement >= relinking_best)
37      {
38        best_relinking_sch = source;        //Best solution
39        best_relinking_nodes = nodes;       //Best nodes
40        relinking_best = improvement;       //Update the best relinking
41        one_improv = true;                  //One improvement is found!
42      }
43      //Move to the next step! Recursion!
44      relinking_phase(source, target, relinking_best, improvement, DEPTH)
        ;
45    }
46    //If an improvement has been found during the relinking, update
47    if(one_improv)
48    {
49      one_improv=false;                           //Improve only once
50      std::swap(nodes,best_relinking_nodes);      //Select the best nodes
51      postprocessing(best_relinking_sch);         //Apply post-processing
52      update_best_schedule(best_relinking_sch); //Update the elite pool
53    }
54  }
```

Listing 1: Relinking process

3.3 Get moves

Getting the legal moves between two schedules is done using the Get_moves function. This function has the task to identify all the possible legal moves that can be applied to the starting solution. Since all the nodes can activate any type of VM, we first need to

eliminate any type of symmetry between the two schedules. As the number of the node do not affect in any way the time required for the training of a job, two Setup will be deemed to be different only if they differ by the VM type or by number of GPU used. Additionally, since a VM can host multiple jobs at once, we actually need to asses if the guiding setup can be accommodated in the starting schedule, without violating any rule of the system, i.e, setting multiple VM on the same node. This is done trough a function called "Compatible" that will search for a node that can actually fit the move. If a node is found, the move is legal and will be added to the list, otherwise it's discarded and the algorithm continues. This function only consider the possibility of changing a single setup per move, it can't perform the swap of multiple setups inside the starting schedule. This possibility was explored during the development of the code, but performing the swap often caused little to no change in the final cost, while it was burdensome from the computational point of view.

```
std::list<std::pair<Job, int>>
Path_relinking::get_moves(job_schedule_t &source, const job_schedule_t
    &target)
{
  std::list<std::pair<Job, int>> c_moves = {};
  int node_idx = {};  //Node idx

  for(Job& j: submitted_jobs){
    if (!target.at(j).isEmpty())
    {
      //Setups must be different
      if (!same_setup(source.at(j), target.at(j))){
        //Search for a node that can accomodate the setup
        node_idx = compatible(source, j, target.at(j));
        if (node_idx!=-1){
          //Add move to the pool
          c_moves.push_back({j, node_idx});
        };
      };
    }
    //There is always space to remove a Setup
    else if(!source.at(j).isEmpty() && target.at(j).isEmpty()){
      c_moves.push_back({j, -1}); //-1 is just a placeholder
    }
  };
  //Print the moves in the output file (for deubbing)
  //print_moves(source, target, moves);
  return c_moves;
};
```

Listing 2: Move generator

3.4 Explore step

The exploration part of the algorithm is done trough the function "explore_step". This function will take as inputs the source and the target schedule, the move, and a bool that

will be used for recursion. The first part is dedicated to freeing the resources occupied by the old setup and allocating the resources required by the incoming setup. Then, the map that represent the solution is updated and the difference between the old and the new proxy function is calculated. The difference is used as a metric since it only requires the computation of the costs of the old and the new setup, saving us from iterating trough the whole schedule each time we need to explore the effects of a move. To improve the efficiency of the relinking, a recursion has been inserted in this part of the code. The modified schedule will be used to calculate a new set of moves, and their effects will be explored using the explore step function again. This recursion is important because it allow to check for moves that are seemingly bad, but allow to access previously forbidden scenarios, that may lead us to better solutions. Once the value of the step has been determined, the schedule is reverted back to normal and the exploration phase can continue. If the exploration phase has ended, and we actually want to apply the move permanently, this last part of the function is ignored, and the modified schedule is now used a the starting one for the next step.

```
1  double
2  Path_relinking::explore_step(job_schedule_t &source, const
       job_schedule_t &target, const std::pair<Job, int> &move, bool
       FORESIGHT, bool permanent){
3
4    Job j = move.first;                  //Job to be modified
5    Schedule ins = target.at(j);         //Setup to insert
6    Schedule rem = source.at(j);         //Setup to remove
7
8    unsigned node_idx = 0;
9    unsigned GPU_source = 0;
10
11   unsigned new_node_idx = move.second;
12   int GPU_target = 0;
13   unsigned GPU_rem = 0;
14
15   //Remove the old setup
16   if (!rem.isEmpty())
17   {
18     GPU_source = source[j].get_setup().get_nGPUs();
19     node_idx = source[j].get_node_idx();
20     GPU_rem = nodes[node_idx].get_usedGPUs();
21     nodes[node_idx].set_remainingGPUs(-GPU_source);
22
23     //If there are no more jobs in the node, we close it
24     if (nodes[node_idx].get_usedGPUs() == 0){
25       nodes[node_idx].close_node();
26     }
27   }
28
29   //Insert the new setup
30   if (!ins.isEmpty())
31   {
32     GPU_target = ins.get_setup().get_nGPUs();
```

```
33      if (!nodes[new_node_idx].open()){
34        nodes[new_node_idx].open_node(ins.get_setup());
35      }
36      nodes[new_node_idx].set_remainingGPUs(GPU_target);
37      ins.set_node(new_node_idx);
38    }
39
40    //Modify the Schedule
41    source[j] = ins;
42
43    //Check what is the difference in old and new proxy function
44    double cost = update_best_cost(j,ins,rem, GPU_rem);
45
46    if(FORESIGHT){
47      //Calculate the new moves
48      pool moves_FS = get_moves(source, target);
49      //Remove moves that were already available before
50      for (auto it = moves.begin(); it != moves.end(); it++){
51        moves_FS.remove(*it);
52      }
53      //The base has to stay the same each iteration
54      double baseline = cost;
55      for (auto move : moves_FS){
56        //Explore one step ahead
57        double price = baseline + explore_step(source, target, move,
     false);
58        if (price > cost){
59          cost = price; //Update the cost of the move with the best one
60        }
61      }
62    }
63
64    //Revert the situation to the previous one if not permanent
65    if (!permanent){
66      //Guiding setup is removes
67      if(!ins.isEmpty()){
68        nodes[new_node_idx].set_remainingGPUs(-GPU_target);
69        if (nodes[new_node_idx].get_usedGPUs() == 0){
70          nodes[new_node_idx].close_node();
71        }
72      }
73
74      //Starting setup is inserted
75      if (!rem.isEmpty()){
76        if (!nodes[node_idx].open()){
77          nodes[node_idx].open_node(rem.get_setup());
78        }
79        nodes[node_idx].set_remainingGPUs(GPU_source);
80      }
81      //Schedule is reverted back to normal
82      source[j] = rem;
83    }
```

```
84
85    //Cost is returned
86    return cost;
87 }
```

Listing 3: Explore move

4 Tests

To perform the tests, the first step is to choose the size of the server farm and other variables like the total number of jobs and the modes of arrival. Some default values are already present, but they can be changed by modifying the content of the "config.ini" file in the main folder. Once these parameters have been decided, use "make data" to generate a csv that will simulate the job submission to the server. The simulations can now begin by using "make tests". This step can be parallelized by adding additional arguments to this make: using "make tests ARGS="-j4" " will parallelize the tests on 4 cores for example. The results of the simulations will be stored in "build/data/tests_new". This folder contains the various schedules generated by the algorithm, and other useful files like the cost over time and the tardiness associated with each job. To automatically collect and compare the costs of the various heuristic methods, use "make analysis". This will generate a csv file inside the main folder called "results.csv", where the results from the available methods are compared. This command will also generate some useful graphs, stored in the "Plots" folder. More details about possible problems and how to read the results are given in the README.md file.

5 Results

5.1 System loads and Data analysis

Before discussing the results, it's important to make some analysis about the submissions regime that have been simulated. One of the most important parameters for the simulation is in fact the mean submission time (MST) of a job to the server farm, as it will dictate the length of the job queue and weather or not resource management is fundamental. During the various tests, three different types of submission regimes have been analyzed: exponential, high and low. They are characterized by a MST, lower, equal or higher than the mean time required to complete a job, scaled by the system size. In the exponential regime, the jobs are bounded to violate the deadlines, as most of the them are actually sent to the server in the first part of the simulation. Being able to fit multiple jobs inside a schedule is fundamental. Both random greedy and path relinking should be used in this regime, as the solution for this type of problem is far from the greedy one. In the high regime, the MST is equal to the mean time to complete a job. Almost all the nodes are always open during this simulation, and resource management is still fundamental, since a miss-management could actually send the whole job queue

into tardiness, violating the various deadlines. The low regime is not very important for the type of problem we are trying to solve, since respecting the deadlines is trivial: every job can be trained with its most cost efficient setup by using the solution offered by the greedy algorithm. Most of the nodes during this simulation are in fact closed, resources far exceed the need of the job queue. This is no longer a job scheduling problem, but a packing one. Path relinking and random greedy are actually able to find a better solutions than the one proposed by the greedy algorithm, since they will rearrange the jobs in the nodes to benefit the most from any idle GPU left in the node by the greedy process. To better understand and visualize the effect of the various submission modes on the server, a python script "compute_nodes.py" has been developed in this project, and its output is reported in *fig* 2.

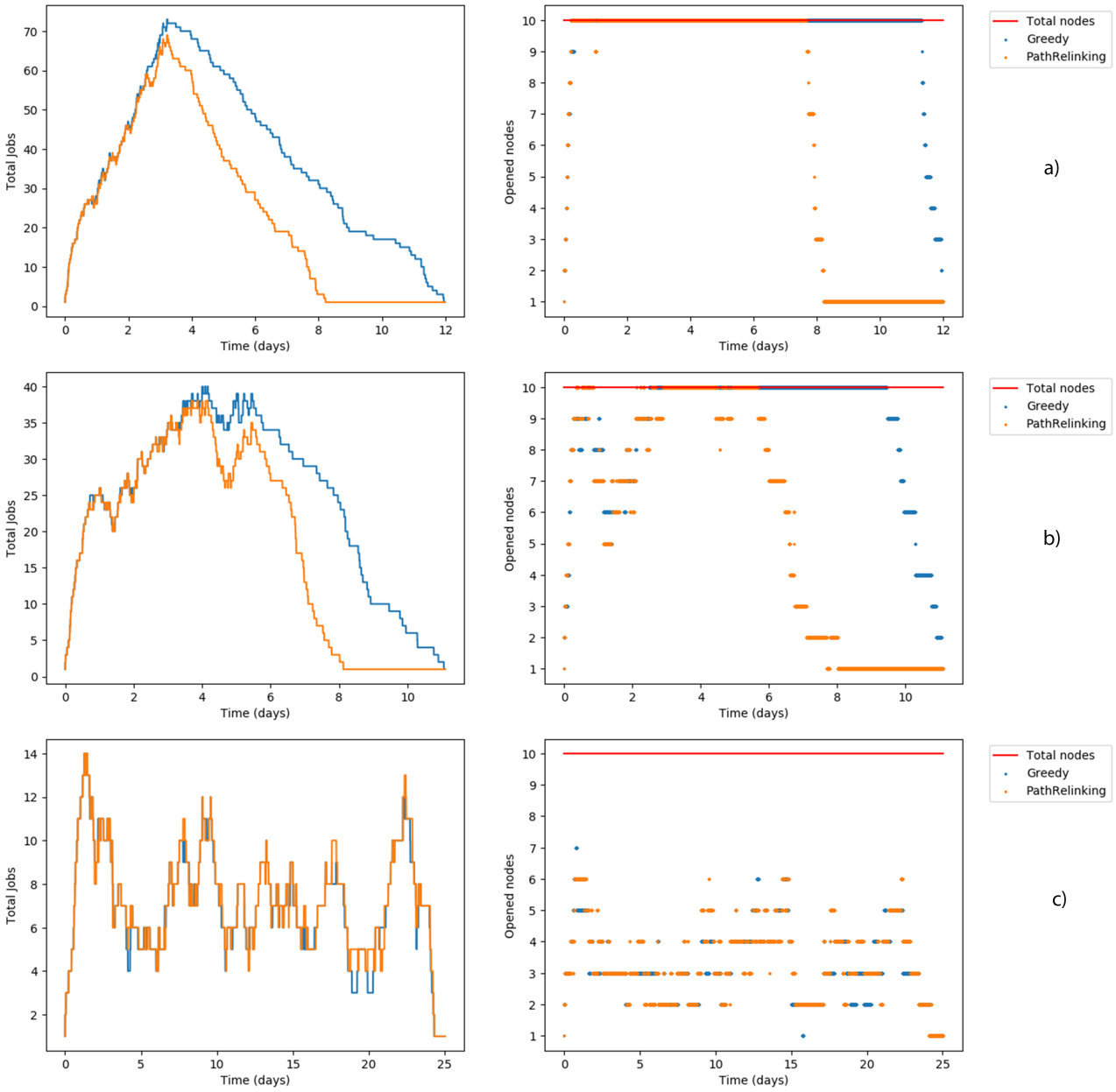

Figure 2: Graphs produced by the python script. a: results of the exponential submission mode, the nodes are always open and the job queue increase until all the jobs have been submitted. b: results of high submission mode, the nodes are almost always open, but the length of the job queue no longer grows until all the jobs are submitted. c: results of the low submission mode, most of the nodes are closed during the simulation

5.2 General results

The results of the simulations are reported in fig 3. The system size was varied from 10 to 50 nodes, and each of the random heuristic method performed 1000 iteration per scheduling operation. Additionally, since we are using random methods, each of the costs reported by random greedy and path relinking is actually the average over 10 simulations using different seeds. Path relinking is shown to almost always perform better than the two other heuristic methods. Most of the improvement are in the exponential scenario, where the cost of the simulation is mainly dictated by the penalties corresponding to the deadlines violation. Path relinking correctly manages to fit multiple jobs inside the schedule, avoiding violating these deadlines as much as possible. For high scenarios, usually the gain offered by both random greedy and path relinking is minimal, below 10%, since the greedy solution is enough to guarantee to respect the various deadlines. For small systems however, due to lack of flexibility, sometimes the simulations actually violate the deadlines, and using the two other heuristic method can drastically reduce the costs. In low scenarios, the gain reported by path relinking is now related to a better reorganization of the jobs inside the nodes, allowing to use idle GPUs in the most cost efficient manner. It is also important to underline that sometimes Random Greedy can actually perform better than Path relinking in high load scenario. This possibility is extremely small, but sometimes using the most cost efficient setup means executing the jobs for a longer period of time, and this may cause problems if multiple jobs are sent to the server in the next time step. Nonetheless, the price difference between path relinking and random greedy should still be minimal.

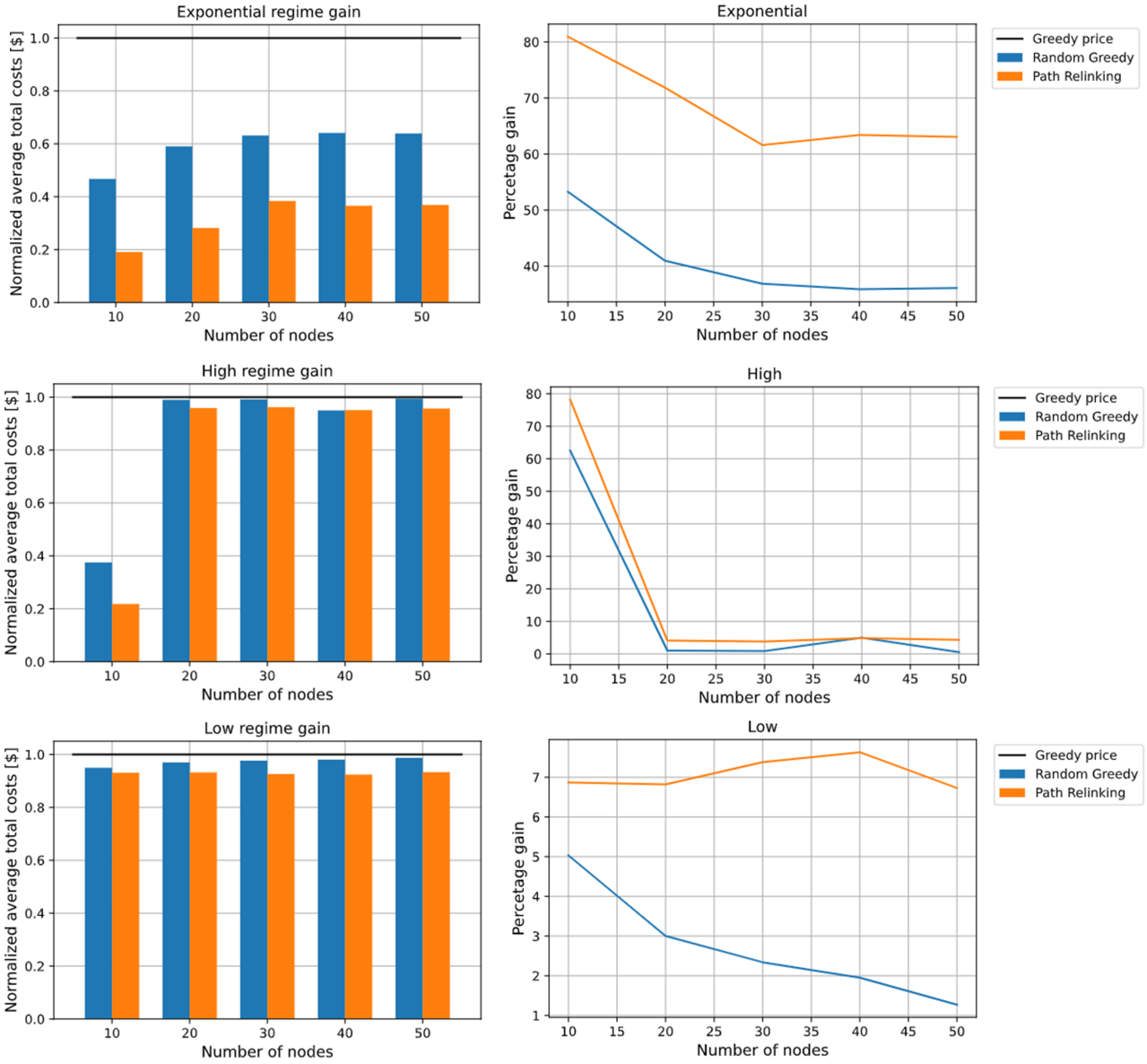

Figure 3: Results of the various simulations. In the exponential regime we see a major gain with respect to the greedy algorithm. In low and high regimes path relinking still perform better than the other two heuristic methods, but the gain is reduced

5.3 Path relinking overhead

While path relinking performs better than the pure randomized greedy procedure, it should be noted that the overhead caused by the algorithm is significant, especially for large systems. A graph reporting the mean execution time is reported in fig 4. Even

after all the modifications performed to speed up the process, the algorithm still scale poorly with the system size, due to the larger number of paths the relinking has has to explore. This overhead can however be somewhat tuned, by changing the algorithm parameters. Reducing the number of elite solutions, or the number of steps that the algorithm can make will reduce the overhead caused by the relinking process for example. Currently, at 50 nodes, the time required to compute the resource scheduling is still extremely small, around 0.5 seconds, for both exponential and high submission regimes. This actually means that a solution where path relinking is used, is still feasible. The same however cannot be said for extremely large systems, that haven't been explored during the development of this project. As the system size increases, the time interval between two submission shrinks, while the time required to schedule the resources is increased. Tuning the path relinking parameters (as well as the random greedy ones) could be object of future studies.

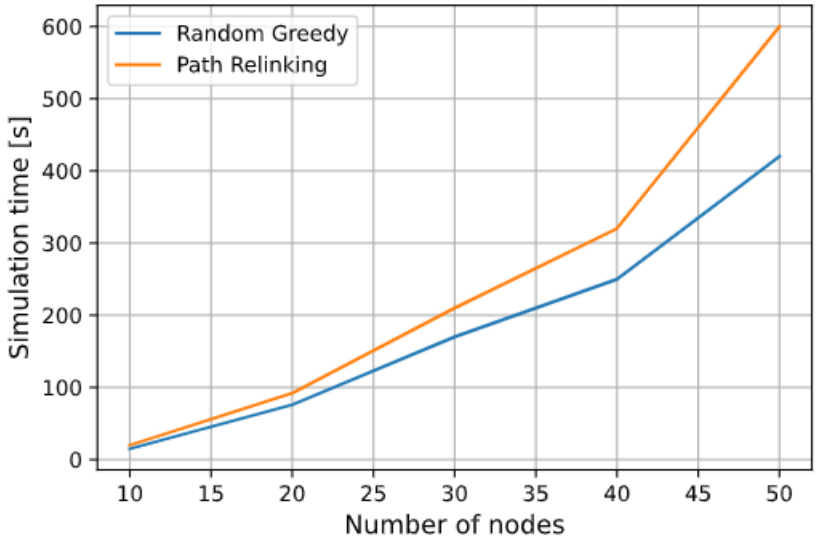

Figure 4: Total simulation times of both random greedy and path relinking with respect to the system size in exponential scenario, 1000 iterations each scheduling

5.4 Validation and other changes

To make sure that the results produced by the script are actually due to an improvement in the scheduling, and not from a bug in the code, a python script that validate the results has been created. This script is in notebook form, and compare the price of each job generated by the Path Relinking schedule and its minimum theoretical cost. Needless to say, if the theoretical minimum price exceed the Path Relinking one, something is fundamentally wrong with the code. This validation of course do not assure of having a bug-less code, but at least the costs prediction do not violate any theoretical limit. Another improvement to the code, not strictly related to the path relinking, is the speedup of the exponential scenario. Compiling the program with gprof showed that most of the

time is spent by creating objects of the class Dstar. This class is fundamental to determine what resources the job can receive to be completed in time. However in very high load scenario, like the exponential one, resources are depleted way before reaching the end of the queue. This means that creating an object of this class is meaningless for most of the jobs, as they cannot receive resources anyway. For this reason, a simple function to check if resources are still available has been created. This function is reported below, and if it return false, will avoid searching for suitable setups for the remaining jobs in the queue. The algorithm is now at least two times faster in this scenarios, while little to no overhead is introduced in other submission modes.

```
bool
Random_greedy::check_resources(){

  //We start by checking from the last node in the vector, as it is
  //The least likely to be occupied/opened, just by following the
  //logic of the program (it opens the nodes in order)
  std::vector<Node>::reverse_iterator node_rit = nodes.rbegin();

  for(; node_rit!=nodes.rend(); node_rit++){
    Node& node = *node_rit;
    //Check if the node is open
    if(node.open()){
      if(node.get_remainingGPUs()!=0){
        //GPUs are still available
        return true;
      }
    }
    //If the node is closed, we have resources
    else{
      return true;
    }
  }
  //All resources have been taken
  return false;
};
```

Listing 4: Resource check

6 Conclusions

Path relinking successfully generated better results than just using the randomized greedy approach in all the considered scenarios. While the overhead caused by this intensification strategy is significant, the overall results justify using it. It is also important to underline that situations analyzed are limited to using only 2 GPU types. If the solution space is expanded, by increasing the GPU types available, path relinking is bounded to yield even better results. Additionally, the overhead caused by the relinking process should be almost independent from the types of GPUs that the system can offer, while the same cannot be said about the GRASP, that needs to explore every type of Setup possible.

References

[1] Federica Filippini Marco Lattuada Edoardo Amaldi, Danilo Ardagna. Job scheduling and optimal capacity allocation problems for deep learning training jobs. Master's thesis, Politecnico di Milano, 04 2020.

[2] Mauricio Resende and Celso Ribeiro. Grasp with path-relinking: Recent advances and applications. *Operations Research/ Computer Science Interfaces Series*, 32, 01 2005.

[3] Mauricio Resende and Celso Ribeiro. *Greedy Randomized Adaptive Search Procedures: Advances, Hybridizations, and Applications*, volume 146, pages 283–319. 09 2010.