Executive Summary of the Thesis

A stochastic approach for scheduling AI training jobs in GPU-based systems

Laurea Magistrale in Mathematical Engineering - Ingegneria Matematica

Author: Lorenzo Marchi
Advisor: Prof. Danilo Ardagna
Co-advisor: Federica Filippini
Academic year: 2021-2022

1. Introduction

Deep Learning methods are currently used to address a variety of complex tasks. This is partially motivated by the fact that these models are now trained on GPUs, expanding the range of problems that can be solved in a reasonable computing time. This has caused the demand for high-performance GPU-based cloud servers to increase dramatically, making it necessary for Cloud Service Providers (CSPs) to manage that demand effectively. In this thesis, we optimize the scheduling of Deep Learning training jobs from the perspective of a CSP running a data center, efficiently selecting resources for the execution of each job in order to minimize average power consumption.

2. Problem Statement

The scenario we want to model consists of two components: first, the cluster, composed of several machines, each with different numbers and types of GPUs; second, the list of submitted jobs, each associated with an execution deadline and a priority for fulfilling it. The aim is to decide which type and how many resources to allocate for the execution of each job, trying to meet the due dates but minimizing the power consumption costs. The model of our optimization problem is extended from [3], by introducing GPU sharing and stochastic profiles of job execution. For each submitted job, the scheduler decides whether to run it or postpone it. A suitable configuration is then selected for each of the executed jobs, choosing the number and type of GPUs. It is possible to partition the GPUs on a node in order to run multiple jobs at the same time. Indeed, the interference generated by sharing a node is negligible, as suggested in [3]. When two or more jobs share a single GPU, this is not true anymore; nevertheless, we decided to consider this scenario. Indeed, nowadays GPUs have a large amount of memory, so multiple mini-jobs can be executed simultaneously, freeing resources for more demanding jobs. To consider the performance degradation, we have to introduce a coefficient to inflate the training time of every job co-located in a single GPU. Given these possibilities, the optimization problem consists in minimizing the cost given by the power consumption of the used nodes and GPUs, plus artificial penalties for violating deadlines and postponing jobs. Since our goal is, when possible, to end the training on time, we
set the tardiness cost to be an order of magnitude higher than the consumption cost. In order to decide which resources to allocate to each job to meet deadlines and minimize costs, it is necessary to estimate the time required for the execution based on the used configuration. We resort to Machine Learning models, as done in [3], to obtain the time required to reach the maximum number of epochs. However, a job may require less epochs to end its training. So we also infer the probability of ending after a certain number of epochs, defining stochastic execution profiles from it. This problem is tackled in an online scenario: a new job can be submitted at any time. Whenever this happens, we re-optimize the previous solution since it could be necessary to free some resources already assigned to other jobs with lower priority than the new one. In this regard, migration and preemption are permitted, following other literature proposals (e.g., [3]). That is, jobs can be interrupted, postponed, and then resumed later; moreover, they can be moved to other nodes with a different number or type of GPUs. It is important to note that we do not have any information about the future; we cannot predict when and which jobs will be submitted. It is, therefore, impossible to find a schedule that will optimize the training of all jobs in the long run. Since we cannot characterize the system throughout its lifespan, we are constrained to find the local minimum of the problem with the information up to that point. A local solution obtained in this way could adversely affect the overall schedule. However, this limitation is due to the intrinsic nature of the problem itself. In summary, we set a rescheduling point whenever a new job is submitted or an already given job completes. Then, we find a new solution to the problem, considering the new information and redefining the old schedule altogether. We modeled this problem through a Mixed Integer Linear Programming stochastic formulation extending the work present in [2] with GPU sharing and stochastic training profile. In this Executive Summary, we report only the proxy function of our formulation and a brief mention of the notation necessary for its explanation. The data center is modeled through a set \( N \) of available nodes, while the set \( J \) denotes the submitted jobs. The decision variables of our formulation represent the assigned resources in terms of selected node \( n \in N \) and number of GPUs assigned to each job \( j \in J \) (or its postponement). Given the selected configuration for \( j \), we know its expected execution time, which is a random variable because it depends on the probability distribution of ending after a certain number of epochs. The objective function of our model is:

\[
\min \mathbb{E} \left[ \sum_{j \in J} \left( \omega_j \tau_j + \rho \omega_j \tilde{\tau}_j \right) + \sum_{j \in J, n \in N} \alpha_{jn} \pi_{jn} \right]
\]

The total cost of a selected configuration includes the contribution of three different terms. The first one is the sum over \( j \) of the product \( \omega_j \tau_j \). The variable \( \omega_j \) represents the tardiness weight for \( j \), i.e., a unit time cost that models the priority for fulfilling the due date. So, being \( \tau_j \) the difference between the expected execution time of \( j \) and its deadline, this term evaluates the tardiness cost for all the executed jobs. Since \( \omega_j \) is an order of magnitude higher than the power consumption cost, this term can significantly impact the objective function. In this way, we force the system to avoid violating deadlines, especially for jobs that have higher priority. The second term is analogous to the first one, but, instead of \( \tau_j \), we consider \( \tilde{\tau}_j \), which is equal to the maximum execution time of \( j \) if it is not executed, and is 0 otherwise. Thus, with this term we penalize job postponement. A postponed job that might violate the deadline can still be completed without tardiness if, in a subsequent rescheduling, it is executed with the necessary resources. However, having no information about future job submissions, the best strategy from the local point of view is to prepare for the worst-case scenario by trying to run as many jobs as possible. To further penalize jobs postponement, we added the parameter \( \rho > 1 \). Finally, the last term models the execution cost. For every node \( n \in N \), we select the first ending job with \( \alpha_{jn} \), and we multiply it to \( \pi_{jn} \), i.e., the cost of using node \( n \) for the duration of the execution of \( j \). The first-ending job time is used because we set a rescheduling point whenever a job is completed, reprocess the system, and possibly change resource assignments. Since all these terms depend on the execution time, the total cost thus obtained is a random variable, so we minimize its expected value.
3. Solution

The problem presented in the previous section cannot be tackled directly. Its online nature requires obtaining the solution at each rescheduling time in a few seconds. Since our mathematical formulation depends on a random variable, it must be solved with a stochastic approach, which inevitably increases the level of complexity. Moreover, it includes many variables and constraints, so it is impossible to solve even a single instance of it by relying on state-of-the-art solvers in a short time. So, we resort to a heuristic method that can provide good-quality solutions in an adequate time for our framework.

The heuristic we propose is divided into three steps: preprocessing, optimization and assignment, and postprocessing. At every rescheduling point, we define the list of submitted jobs $J$, virtually merging the list of the ones already in execution, the previously postponed ones, and the ones just submitted. In the preprocessing phase, for each $j \in J$, we compute/update a pressure index based on some parameters like the deadline $d_j$, the tardiness weight $\omega_j$, or the minimum execution time $t_{\text{min},j}$. This index is a real number that models the priority of job execution and is needed to define the ordering used in the optimization step. Choosing an appropriate pressure index is crucial, as the highest a job is in the ordered list, the highest the probability of it being executed, especially in heavy-load situations, when the available resources are not enough and it is necessary to postpone some jobs. The pressure proposed in [2, 3] is:

$$\Delta_j = T_c + t_{\text{min},j} - d_j,$$

where $T_c$ is the current time. It allows to consider the proximity of job $j$ to the deadline based on its training speed. For our heuristic, we adopted a pressure index equal to $\Delta_j$, except for the case $\Delta_j > 0$, i.e., when $j$ will violate the due date, where it becomes $\omega_j \Delta_j$. This way, we take into account the intrinsic priority of the job. In the second step, for each job, an optimal configuration is selected; if there are available resources, we assign them to the job; otherwise, we try sub-optimal configurations. When possible, job allocation is performed by assigning the GPUs of an already active node rather than opening another one. The same argument holds for a GPU fraction: first, we look for already shared GPUs, ordering them by the remaining available space. We stop when all nodes are full, or there are no more jobs to be assigned. The idea behind the optimization step is to find optimal configurations using the stochastic information for the training time as made in [1]. In practice, we exploit the mathematical model proposed in this paper to obtain a stochastic schedule for each submitted job. This consists of a dynamic schedule with different configurations, assigned to the job as the training proceeds. Specific points in time are identified such that, when the job execution reaches them, the number of GPUs assigned to it is changed. The objective of this updating configuration is to minimize the average energy consumption by exploiting the probability that a job terminates before the maximum number of epochs. Frequently, a DL job terminates if the accuracy improvement remains below a certain threshold for a given number of subsequent epochs. In any case, if a job has a slow improvement rate that does not fall below the threshold and is terminated based on the maximum number of iterations, the schedule ensures that the deadline is still met. As in [1], the general idea is that starting with a low-power configuration and then gradually increasing the training rate can reduce the cost of a job finishing early and can fulfill the due date otherwise. The stochastic model in [1] is designed for a simplified scenario and needs to be adapted to our more complex problem. First, it considers only one job to be executed in a single node with fixed resources exclusively dedicated to it. The adaptation, in this case, is simple: although we have multiple jobs to schedule, the structure of our heuristic is such that they are ordered and considered individually. In this way, we can use the model as it is for each job. Another issue is that our problem has an online nature and changes continuously over time. Therefore, the stochastic model has to be translated in time. Furthermore, it is impossible to guarantee that the stochastic schedule assigned to each job will be feasible, because new submissions may make it impossible to guarantee the scheduled resources. Having no information about the system future and knowing that some jobs will run with a sub-optimal configuration, it is worth considering only a local solution to the stochastic problem and reevaluating it when-
ever necessary. Therefore, although solving the stochastic problem provided a complete execution profile, we only consider the starting configuration and then compute a new solution at each rescheduling point. Note that, with this approach, the rescheduling of the whole optimization is not given anymore only by the submission of a new job or the end of a job currently in execution: we have to consider also the time $t_j$ until the next assigned resources increase to job $j$. In this way, when $t_j$ time has elapsed, the stochastic profile of job $j$ is updated, and we assign to it the next configuration. We solve the instance thus generated using an interior point method, which gives us optimal solutions since the optimization problem is convex and the objective function is differentiable. Finally, in the postprocessing step, we check if the shared GPUs are full; otherwise, we try to assign a higher fraction of the entire GPU to the shared jobs to limit the amount of idle resources. Moreover, if the type and number of GPUs assigned to a job did not change from the previous schedule, an attempt is made to assign the job to the same node, avoiding unnecessary migrations. The scheme of our heuristic is reported in Algorithm 1. First, on line 3, we get a sorted list of jobs $J_s$ by ordering the list of submitted jobs $J$ by the pressure index $\Delta$ described before. We iterate over this list and, for every job, we search for optimal configurations $D_j$ on line 6, solving the stochastic model with the aforementioned modifications. This is done for all jobs until there are none left in the list of submitted ones or there are no more available resources, in which case we place the remaining jobs in the list of postponed ones. Finally, on line 16, we perform the postprocessing step to assign idle resources in shared GPUs and leave already executing jobs in the same nodes whenever possible.

4. Experimental Results

To verify the effectiveness and generality of our heuristic method, we generated a set of random problem instances using the parameters described in the following. We set the number $N$ of available nodes in the system varying from 10 to 100 with a step of 10, selecting from every possible scenario $10N$ jobs. We generate the system nodes starting from a GPU catalog of Nvidia K80 and M60. For inter-arrival job generation, we used three different distributions (low, high and exponential) as proposed in [3], obtaining a wide range of scenarios from low to high inter-arrival frequency; this way, the results can be validated with different workloads. Three problem instances are generated for each value of cluster size and each possible inter-arrival distribution, varying the random seed. In order to evaluate our stochastic heuristics, it was necessary to model the probability distributions for the number of epochs required to complete the training of a Deep Learning job. For this purpose, we collected data on the execution of DL tasks of various kinds using different neural networks and AI frameworks on the server of AIR-Lab at Politecnico di Milano. Given these experimental settings, we performed numerical simulations to test the quality of our heuristic.

4.1. Comparison Results

The methods we used to compare against our stochastic heuristic (STS) are the Random Greedy (RG) heuristic, presented in [2], and Earliest Deadline First (EDF), a simple first-principle method that orders the jobs based on their deadline. We launched the simulations and collected the results for each cluster size and inter-arrival distribution, considering the average execution cost for all random instances. In Figure 1, we show the average total cost ratio of all three methods compared with EDF. In each scenario, it is evident how our method with the stochastic schedule performs significantly better than the other two, achieving a remarkably lower energy cost. The average percentage cost reduction $\text{pcr} = (c_{\text{other}} - c_{\text{STS}})/c_{\text{other}}$ is around
Executive summary

Lorenzo Marchi

38% against the RG method and 40% with respect to EDF. Moreover, in every simulation, the STS method succeeds in avoiding tardiness as the RG, demonstrating a conservative nature of the stochastic approach. Another heuristic, more efficient than RG, was developed in [3], namely the Path Relinking. In the simulations authors carried out, the pcr of the PR compared to the RG was about 10%, so our method should perform better than the latter. The reason for such a significant energy cost reduction is undoubtedly because RG and EDF, unlike our STS, use only worst-case job execution times and have no stochastic information about their training. To further test the quality of our heuristic, we ran the same simulations with RG and EDF, this time providing them with the average execution time instead of the worst-case one. Although in this scenario the energy cost is lower than that obtained with our heuristic, we get a more significant contribution from the tardiness cost. Overall, the total cost obtained by RG is similar to the worst-case version, while we observe a marked degradation for EDF, which has low energy costs but extensively violates the deadlines. This is because, considering the average execution time, RG and EDF assign minimum-cost configurations to the jobs; if the execution ends earlier than average, they get excellent results; otherwise, having to balance the slow initial training, frequently they are not able to recover the delay, and the job is led into tardiness.

4.2. GPU sharing

In our work, we considered the possibility of running multiple shared jobs in the same GPU. We enabled this scenario in simulations to evaluate which energy cost reduction it can yield compared to the standard case. The execution time for jobs co-located in the same GPU is higher than in the case of an exclusively allocated GPU. In order to properly model this performance decrease, we multiply the standard execution time on one GPU by an inflate parameter. This parameter is randomly selected from a uniform distribution between 1.01 (1% increase) and an inflate_max parameter that depends on the assigned fraction. To evaluate the gain in energy cost, we launched simulations with our heuristic enabling and disabling GPU sharing. In Figure 2, we report the average percentage cost reduction obtained for every considered system size. It can be noticed that, allowing to run multiple jobs on the same GPU induces a significant reduction in energy costs. The percentage reduction in cost $pcr = (c - c_{sh})/c$ is around 17.6% on average for the exponential case, 17.9% for the high case, and 16.8% for the low case.

4.3. Scalability analysis

In addition to analyzing the quality of the obtained solutions, another critical factor in evaluating the performance of our heuristic is the average time needed to solve a single instance of the problem. We inspected the performed simulations to determine the maximum number of concurrent jobs at a rescheduling point of our problem. For systems of 100 nodes and 1000 jobs, this value is about 400 in the high-rate scenario, 200 for the exponential inter-arrival scenario, and 100 for the low-rate scenario. Thus, we solved a single instance of the problem with our heuristic by generating a number of jobs equal to $4N$, $2N$, or $N$, where $N$ is the number of nodes, submitting them all at the initial instant. The results are shown in Figure 3. From the plots, we observe that the average execution time of a single instance has a linear dependency

Figure 1: Average total costs obtained with all methods
Executive summary
Lorenzo Marchi

Figure 2: Percentage cost reduction obtained with GPU sharing

Figure 3: Average execution times needed for STS to solve a single instance of the problem

on the number of jobs to be trained concurrently, and is always under a minute, even for the larger systems.

5. Conclusions

In this thesis, we presented a stochastic approach to model and tackle the optimal scheduling problem for AI training jobs in GPU-based systems. We set up an extensive experimental campaign and performed simulations to test the quality of our method. The results confirm that our heuristic guarantees significantly better results than the existing ones, with a percentage reduction in energy cost of about $38 - 40\%$. In order to assess the benefit of GPU sharing, we also ran simulations and compared the power consumption cost when this was or was not allowed. The results demonstrate that, depending on the workload and GPU memory, the possibility of co-locating multiple jobs on a single GPU can reduce the cost percentage between 17\% and 29\%. Finally, by showing that solving instances of the problem with up to 100 nodes and 400 concurrent jobs requires less than 60 seconds, we demonstrated that our method could be applied efficiently in real-world scenarios.

References

