READYS: A Reinforcement Learning Based Strategy for Heterogeneous Dynamic Scheduling

Nathan Grinsztajn
Univ. Lille
CNRS UMR 9189 CRIStAL, Inria
Lille, France
Nathan.Grinsztajn@inria.fr

Olivier Beaumont
Inria Bordeaux Sud-Ouest
Univ. Bordeaux
Bordeaux, France
Olivier.Beaumont@inria.fr

Emmanuel Jeannot
Inria Bordeaux Sud-Ouest
Univ. Bordeaux
Bordeaux, France
Emmanuel.Jeannot@inria.fr

Philippe Preux
Univ. Lille
CNRS UMR 9189 CRIStAL, Inria
Lille, France
Philippe.Preux@inria.fr

Abstract*—In this paper, we propose READYS, a reinforcement learning algorithm for the dynamic scheduling of computations modeled as a Directed Acyclic Graph (DAGs). Our goal is to develop a scheduling algorithm in which allocation and scheduling decisions are made at runtime, based on the state of the system, as performed in runtime systems such as StarPU or ParSEC. Reinforcement Learning is a natural candidate to achieve this task, since its general principle is to build step by step a strategy that, given the state of the system (the state of the resources and a view of the ready tasks and their successors in our case), makes a decision to optimize a global criterion. Moreover, the use of Reinforcement Learning is natural in a context where the duration of tasks (and communications) is stochastic. We propose READYS that combines Graph Convolutional Networks (GCN) with an Actor-Critic Algorithm (A2C): it builds an adaptive representation of the scheduling problem on the fly and learns a scheduling strategy, aiming at minimizing the makespan. A crucial point is that READYS builds a general scheduling strategy which is neither limited to only one specific application or task graph nor one particular problem size, and that can be used to schedule any DAG. We focus on different types of task graphs originating from linear algebra factorization kernels (CHOLESKY, LU, QR) and we consider heterogeneous platforms made of a few CPUs and GPUs. We first propose to analyze the performance of READYS when learning is performed on a given (platform, kernel, problem size) combination. Using simulations, we show that the scheduling agent obtains performances very similar or even superior to algorithms from the literature, and that it is especially powerful when the scheduling environment contains a lot of uncertainty. We additionally demonstrate that our agent exhibits very promising generalization capabilities. To the best of our knowledge, this is the first paper which shows that reinforcement learning can really be used for dynamic DAG scheduling on heterogeneous resources.

I. INTRODUCTION

Modern computer systems contain a variety of resources, interconnected in order to support parallel and distributed computationally intensive applications. Efficiently executing parallel applications on such systems is of key importance in many scientific domains and requires to accurately allocate and schedule the computations onto the available resources.

Directed Acyclic Graph (DAG) is a very useful computational model for describing parallel applications. In a DAG, each node represents a computational task and each directed edge represents a dependency between two tasks. This model allows a good representation of the parallelism between tasks and is used in many contexts. In this paper, we investigate learning algorithms to allocate and schedule tasks dynamically on a set of computing resources. We focus on task graphs originating from linear algebra, and in which each task typically operates on a sub-matrix/tile. The size of the tiles is chosen to achieve good efficiency on both CPUs and GPUs and typically lasts a few tens of milliseconds. This constraint is important because it naturally guides the complexity of the task allocation and scheduling algorithms to be implemented, since the allocation and scheduling decisions are made at runtime.

Task-based runtime systems [8], [16] internally represent the application as a DAG in order to process it on a parallel machine and the runtime system schedules the different tasks onto the available computing resources. The DAG scheduling problem consists in finding the best way of assigning tasks to processing units, so that the task dependencies are fulfilled and the makespan is minimized. In this context, the exact computation duration of tasks and the exact duration of communications are not exactly known, even though we can have good prior estimates: the problem to solve is therefore of stochastic nature. When using static task allocation and scheduling, even with relatively well-known task and communication durations, a drift is observed when the problem size becomes large, which is a source of load imbalance and idle time. This observation is at the heart of the success of dynamic schedulers such as StarPU and ParSEC [8], [16], which rely on less sophisticated scheduling algorithms because decisions are made at runtime and must be very fast, but benefit from a more accurate knowledge of the state of computation and communication resources at the time of making a decision. In practice, dynamic schedulers make task allocation decisions a little in advance of the actual processing of the tasks, which makes it possible to perform the necessary communications as soon as possible, thus ensuring good overlap of communications with computations.

We assume that communications can be overlapped with computations and can therefore be neglected, which is a reasonable assumption at the scale of the computing node consisting of a few CPUs and GPUs (see Section III). This allows to simplify the problem formulation while keeping its inherent NP-hardness. Linear algebra kernels offer a rich context in which the combined use of CPUs and GPUs is relevant. Indeed, the different linear algebra kernels involved

in a factorization exhibit very different acceleration rates on the GPUs.

To deal with the inherent stochastic nature of the problem, we propose to investigate the use of Reinforcement Learning (RL) to design efficient dynamic scheduling strategies. RL is exactly meant to solve this kind of problems, in which a series of decisions, based on the current state of the system, contribute to the achievement of a global objective, the minimization of the makespan in our case. However, the use of RL in an environment with high efficiency and fast decision making requirements is still a challenge. To apply RL, we have to express the problem as a Markov Decision Problem (MDP) in which the objective is to minimize the makespan.

Dealing with graphs in Reinforcement Learning (and machine learning more generally) largely remains an open issue. Here, we represent only a part of the DAG of tasks to be scheduled: this part is a window w sliding over the DAG, so that at any time, the algorithm considers only a relevant part of the DAG, consisting in the tasks that are ready for processing (whose all predecessors have been processed) and their descendants at distance w (w being 1, 2 or 3), in order to efficiently schedule the next tasks. Moreover, it is reasonable to consider that in practice, the whole DAG is not known in advance. Indeed, numerical tests might typically modify dynamically at runtime the shape of the DAG. Moreover, in practice, scheduling and allocation decisions must be fast with respect to the typical duration of tasks, so that it is necessary to restrict the size of the state on which decisions are based.

In [28], we proposed such an RL approach to deal with tiled CHOLESKY factorization in simple homogeneous environments; we showed that in this context an RL approach is competitive with the ASAP heuristic in terms of the makespan, and that it is possible to transfer the knowledge acquired while solving a problem of size T to solving another problem of size T', hence saving the learning time. In this paper, we go much further in our investigation: we apply this RL approach to a set of factorization algorithms, namely CHOLESKY, LU, and QR, which are very common in numerical linear algebra routines, and we extend our results to heterogeneous platforms consisting of both CPUs and GPUs. In this context, we consider a scheduling problem in which resource performances are unrelated [32], i.e. the ratio between the processing time of a task on a GPU and on a CPU depends on the type of the task itself.

Our contributions are as follows. After discussing the related works in Section II, we model dynamic scheduling on heterogeneous platforms as a reinforcement learning problem in Section III and design a suitable RL agent in Section IV, an agent we name READYS. Then, we perform an experimental study considering standard linear algebra kernels in Section V. We compare the performance of READYS with those of classical heuristics in the heterogeneous case, namely the static HEFT [48] and the dynamic MCT [46] heuristics. We show that READYS is competitive with respect to HEFT even when the prediction of task lengths is accurate. When task durations are not exactly known in advance, we show that READYS performs much better. It is worth noting that HEFT relies on complete and accurate knowledge of the DAG, unlike READYS which grounds its decisions on much less information. In Section V-F, we investigate the possibility to transfer the knowledge acquired while solving the problem on a given (matrix) size to another size. Overall, this paper is the first to demonstrate that reinforcement learning is an interesting approach to dynamically schedule tasks, in particular when the characteristics of the tasks and computing resources are not perfectly known or are changing along time. We emphasize that this lack of exact knowledge, though often neglected, is actually the situation faced on real HPC systems.

II. RELATED WORKS

Among combinatorial problems, task scheduling has attracted a lot of research and offers a rich taxonomy [33]. Task scheduling consists in assigning a set of tasks, whose dependencies are represented by a DAG onto a set of computing resources while fulfilling various constraints: a resource cannot execute several tasks at the same time, a task cannot start before its predecessors have been completed, *etc*. This problem is known to be NP-hard in the strong sense [25]. In the literature, several classes of scheduling problems have been studied [17], [26]. In static problems, the DAG is assumed to be completely known in advance while in dynamic problems, part of the DAG is unveiled as the scheduling algorithm progresses. In the homogeneous setting, all resources are assumed to be identical while in the heterogeneous case, resources are different and the same task can have different durations, depending on the resource it is allocated to. Resources are said to be unrelated [17] if the ratio between the processing times for a given task on two different resources depends not only on the resources but also on the task itself. Scheduling problems are often NP-Complete [17] and even simple cases (*e.g.* tasks with no dependencies and two resources) turn out to be NP-Complete [25].

Recently, the use of Reinforcement Learning (RL) to combinatorial optimization has been considered in many papers [1], [21], [34], [40], [51] and a survey has been proposed in [13]. The performance of RL algorithms for combinatorial optimization remains very far from the performance of dedicated heuristics. The main ingredients in these strategies are the graph representation model, the reinforcement algorithm, and the possible use of additional heuristics to help the agent. (*e.g.* graph pruning in [34]). The problems studied are classical NP-Complete problems, such as the Traveling Salesman Problem (TSP), Minimum Vertex Cover (MVC), and the Max-Cut problem. A few works focus on the possible use of RL for task scheduling, such as Placeto, GDP and others [2], [9], [24], [35], [37], [38], [43], [52]. Most of them consider a stochastic context in which tasks arrive sequentially and randomly [9], [31], [36], without using DAG structure. Recently, [43] uses reinforcement learning to guide a genetic algorithm. However, the scheduling strategy is static, and the environment is completely deterministic, which are two important limitations with respect to practical constraints in our target environ-

ment. In [35] the authors present a reinforcement learning approach to map independent parallel jobs onto a parallel machine. Contrary to our context, there are no dependencies between parallel jobs and the goal is rather to minimize the job slowdown. In [38], the authors concentrate on the task mapping task of Deep Neural Network Graphs. Their approach is based on the policy gradient algorithm. [24] tackles the same problem as [38] by modeling it as a Markov decision process and using a reinforcement learning approach called proximate policy optimization [47]. However, both approaches are not suited for transfer learning as the proposed solutions can only improve the mapping of the input problem. In [2], the authors provide a general approach for mapping a task graph, using a graph embedding approach called Placeto. In [52], the authors introduce GDP that uses the same approach as Placeto but outperforms it in their experiments. These two approaches allow transfer learning for new graphs but not for new machines, since the target platform must be the same as the training one. They also compute a fully static schedule, which is not suitable in the case when the duration of the tasks are imperfectly known. In [37], a dynamic scheduling strategy is learned on top of Spark considering online DAG job arrivals. Contrary to our approach, scheduling decision are made at *stage level* (a stage being a set of independent tasks operating on different input data), leaving fine-grained task-level decision to Spark. This eliminates the burden of dealing with large DAG, but restricts the approach to jobs with high inherent parallelism.

Three papers close to our work are [39], [42], and [50]. In [39], the authors consider a very realistic environment (communication time, storage capacity of the nodes, *etc*), while [42] tackles the problem of heterogeneous distributed systems. But contrarily to us, both approaches rely on a hard-coded pre-processing of the DAG into a look-up table, which prevents scalability and generalization. Moreover, [42] uses simple q-learning reinforcement algorithms unable to scale to complex environments nor generalize to unseen instances. As in our approach, deep reinforcement learning is used in [50]. Nevertheless, dynamic online scheduling is not discussed in [50], whereas it is a crucial feature in an environment where the number of tasks and the complex interactions between their processing on different resources make it unrealistic to rely on purely static strategies. Moreover, [50] and [39] pre-process the DAG in a way that is incompatible with a dynamic scheduling setting.

Overall, there is still no generic RL approach that addresses the dynamic scheduling of task graphs to cope with the stochasticity of tasks and allows the transfer of results to new problem sizes.

From the scheduling point of view, a lot of work has been done to efficiently perform linear algebra factorizations in parallel on heterogeneous platforms, because of the practical importance of these kernels. We focus here on works using dynamic runtime scheduling strategies. For instance, the CHOLESKY factorization has been implemented in DAGuE [15], StarPU [7], [20], OmpSs [22], and SuperMatrix [45]. The basic task scheduling strategy consists in (i) analyzing the list of ready tasks (i.e. all tasks whose predecessors have already been processed), (ii) taking the most important task (using a priority system typically based on heuristics such as HEFT [48]), and (iii) placing it on the resource that is likely to complete it as early as possible using estimations as in MCT heuristic [46], given the task cost models on the different resources (and the input data transfer times). HEFT and its variants are the de facto heuristics for static scheduling, while MCT is popular for dynamic scheduling: we will use them as reference algorithms to evaluate the performance of READYS in Section V.

III. MODELS

A. Problem Definition

The scheduling problem can be formalized as follows. We are given a DAG that models the set of tasks to be scheduled. Each vertex corresponds to one task, and each directed edge expresses a dependency between the result of one task and its use by a subsequent task. The target machine is composed of heterogeneous computing units (i.e. CPUs and GPUs). In this paper, we focus on applications in linear algebra; in such applications, a matrix is processing and the set of tasks to be performed is made of a small number (typically 4 in this paper) of kernels. One kernel corresponds to a certain processing performed on a sub-matrix/tile of a given size. The execution time of a kernel depends on the computing unit executing it, so that the acceleration factor on a GPU can be larger for a kernel than for another. This is typically the case in practice, due to the suitability of a given kernel on one particular computing unit, or another. Nevertheless, in practice, due to heating conditions or NUMA effects, the duration of the execution of a given task on a given resource is not constant and the variability in the processing time also depends on the resource on which they are performed [11].

In what follows, we assume that it is possible to overlap communications with computations, so that we can neglect communication costs. This assumption is justified by the possibility to choose the granularity of the tasks, i.e. the size of the tiles. In the case of tiles of order N, the amount of data to transfer is $O(N^2)$ while the complexity of the different kernels is $O(N^3)$, so that one generally chooses N large enough to overlap computations and communications, while allowing an efficient use of computation resources (cache, *etc*). For a matrix of size $M \times M$, there are T^2 tiles where $T = M/N$. In the numerical linear algebra applications considered here, each tile is processed several times so that the number of tasks of the DAG is $n = O(T^3)$. Moreover, dynamic schedulers generally make task allocation decisions on the different resources (i) by minimizing data exchanges by taking into account the placement of input data in the allocation and (ii) by making allocation decisions a little bit in advance, maintaining a queue of tasks on each resource, which allows to start transfers as soon as possible and thus to have all data available on the chosen node at the time of launching the task. For a given allocation and a given schedule, the *makespan* is defined as the

completion time of the last task of the DAG to be processed: it is the quantity that we aim at minimizing in this paper. It is noteworthy that in the heterogeneous case there is neither a notion of a critical path (since we do not know in advance where the different tasks will be allocated) nor of total work (since we do not know in advance the fraction of each task type allocated on each type of resource). Extensions of critical path relying on probabilistic estimates have been proposed in [48] and a generalization of the overall work relying on rational number linear programming has been proposed in [6].

Furthermore, since we are interested in learning a dynamic scheduling strategy, we do not assume that the reinforcement learning-based scheduler knows in advance the entire DAG to be scheduled, nor that it is capable of computing global statistics such as task critical path or overall work. The scheduler is myopic, considering only the tasks ready to be processed and their descendants up to a certain depth (see Section III-B for the precise definition).

B. Reinforcement Learning Formulation

We model the dynamic scheduling problem as a Markov Decision Process which is defined by a quadruple (S, A, P, R) where:

1) S is the set of states,
2) A is the set of actions,
3) P is the transition function: $P_a(s, s') = \mathbb{P}(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability of the transition from state s to state s' under the action a,
4) R is the return function: $R_a(s, s')$ is the expectation of the immediate return obtained while transiting from s to s' with the action a.

Then, solving a Markov decision problem (MDP) consists in finding a policy that optimizes a certain objective function defined on a Markov decision process. Reinforcement learning solves an MDP without any knowledge about neither P nor R.

Let us now define S, A, P, R, the objective function, and the policy used to model and solve the DAG scheduling problem. A **state** contains the necessary information about the system to be able to decide which action is best to perform to optimize the objective function. It is precisely the aim of the algorithm to learn a policy, that is a matching between the state and the optimal action that leads to the optimization of the objective function. Hence, the definition of the state is of paramount importance in reinforcement learning: it should contain all necessary information but not more, so that the mapping is accurate and to keep the decision process fast enough. In a real problem, finding a good definition of the state is part of the problem modeling. Moreover, in a real problem, the "necessary" information may be too cumbersome to deal with efficiently, in which case we need to use an approximate definition of the state which may not contain all the necessary information, but enough information so that the algorithm can learn an accurate mapping between the states and the actions, and do that within a reasonnable amount of time. In our context, the state should contain information about the status of the ready tasks and their descendants up to a certain depth in the DAG, and the status of the computational resources so that the agent can decide which task to schedule next on which resource. This simple formalism can be difficult to handle because a lot of task-processor pairs can be considered. Instead, each time a decision has to be made, we choose at random one available processor, named the "current processor". Regarding the DAG, computing the optimal solution requires exponential time and the knowledge of the whole DAG. For the reasons mentioned above, we consider an approximate representation, where we restrict the information represented in a state to the information about running tasks, ready tasks, and some of their descendants (see Fig. 1): running tasks are those that are currently being processed, ready tasks are those that could be processed but have not yet been assigned to a computing resource and descending tasks are those that are dependent on at least one running or one ready task. The depth of a descending task is defined as the minimum length of any path from a ready or running task to it. The set of descending tasks kept in the state is made of all tasks which depth is less than a given window size w. The choice of the parameter w results from a trade-off between computing time and available information within the state. The state of the resources is represented by a vector containing the type of each computing resource (CPU or GPU) and the estimated time at which it will be available, given the task already running on it.

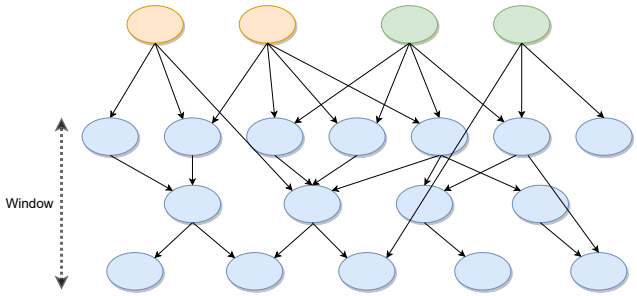

Figure 1. A state contains information about running tasks (orange), ready tasks (green) and their descendants (blue). This plot illustrate these notions: nodes are tasks, edges are dependencies between tasks: a task/node cannot start its execution before all its ancestors have run to completion. In this example, the window w is set to 3.

Each vertex of the DAG, *i.e.* each task, is represented by a set of raw features: these features are expected to encode and summarize the DAG information at this vertex level. We use normalized quantities in order to facilitate policy transfer between graphs of different sizes. The representation $\hat{X}_i$ of Task i can be written as

$$\hat{X}_i = [|\mathcal{S}(i)|, |\mathcal{P}(i)|, type(i), ready(i), F(i)],$$

where $\mathcal{S}(i)$ is the set of immediate successors of vertex i (and $|\mathcal{S}(i)|$ is hence the number of successors of i), $\mathcal{P}(i)$ is the set of immediate predecessors of i (hence $|\mathcal{P}(i)|$ is the number of predecessors of i), $type(i)$ is the type of the task encoded, $ready(i)$ is a binary variable indicating if the task i is ready. $F(i)$ summarizes the information about the descendants of

task i: it is a vector containing the number of descendants of each type normalized by the total number of tasks of each type. More formally, if we denote by 0 the root of the DAG, we can define the unnormalized form $\bar{F}$ of F recursively by

$$\bar{F}(i) = \left(type(i) + \sum_{c \in S(i)} \frac{\bar{F}(c)}{|P(c)|} \right) \text{ and } F(i) = \frac{\bar{F}(i)}{\bar{F}(0)}.$$

A state is represented by such a vector $\hat{X}$. To produce a good schedule for a DAG of tasks, one has to consider not only the current task to schedule, but also the dependencies of the tasks to schedule in the (near) future. Therefore, the state has to combine information about a set of nodes, taking into account the dependencies between them. Obtaining a compact and reliable representation of this information has been a major challenge in machine learning for decades. Recent advances have led to graph convolutional networks (GCN) that, if not perfect, provide a mechanism to obtain such a representation of a graph [30]. GCN is an analog of the convolutional neural networks (CNN) that are applied to images: as CNNs combine the information of neighbouring pixels of an image, GCNs combine the information of neighbouring nodes of a graph. To be more precise, given a representation of the nodes $H^{(l)}$ of a graph G, a GCN layer computes the embedding $H^{(l+1)}$ of each node using local information, according to the formula

$$H^{(l+1)} = \varphi\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right).$$

Here, φ is an activation function, $\tilde{A}$ is the adjacency matrix of the DAG with self-connections added, $\tilde{D}_{i,i} = \sum_j \tilde{A}_{i,j}$, and $W^{(l)}$ is a layer-specific trainable weight matrix. The raw features are used as the starting vector for each node, such that $H^{(0)} = [\hat{X}_1, \hat{X}_2, \ldots, \hat{X}_n]$.

Stacking such GCN layers give rise to a richer representation of the DAG by combining the properties of neighboring vertices [30].

Each time a computational resource becomes available, an **action** consists in selecting an available task to be run on this computational resource, or in staying idle (action $\emptyset$). This $\emptyset$ action makes it possible to implement more complex schedules than list schedules, where for example slow processors are kept idle. The set of possible actions makes A.

The **transition** function P does not need to be explicitly defined and it is not used by the RL algorithm: by itself, the computer system transits from one state to another.

At each instant t, the **return** r_t should provide useful information about the performance of the RL agent with regards to the optimization of its objective, here the makespan. However, it is very difficult to provide relevant information that could be used as an immediate return whereas the whole DAG has not been scheduled yet. Consequently, there is no return (i.e. $r_t = 0$) except once the whole DAG has been scheduled. Hence, we define it as

$$R_a(s, s') = \begin{cases} 0 \text{ if } s' \text{ is non terminal,} \\ R(\text{makespan}) \text{ otherwise.} \end{cases} \quad (1)$$

When reaching the terminal state, the return $R_a(s, s')$ is defined by the makespan of the schedule performed by the RL algorithm normalized by the makespan computed by a baseline algorithm. Here, we use the HEFT heuristic [48] as a baseline. HEFT is a scheduling strategy that assigns the highest priority task (priorities being computed as explained in Section V-C) to the next available resource. If we denote by $makespan$ the makespan achieved by the RL agent and by $makespan(\text{HEFT})$ the makespan achieved by HEFT heuristic, the return is defined as:

$R(\text{makespan}) = \frac{\text{makespan(HEFT)} - \text{makespan}}{\text{makespan(HEFT)}}$.

R is thus positive whenever the reinforcement algorithm is performing better than HEFT.

As already said, an RL agent learns a policy, usually denoted π. π can be represented in various ways, but in any cases, it maps a state to an action. For instance $\pi(s)$ may be a distribution of probability over A, that is the probability to emit each action $a \in A$ in state s; we denote $\pi(a|s)$ the probability to perform action a in state s. As the RL agent learns, π is updated. Though we hope that an optimal policy will be learned eventually (asymptotically), in complex problems like the one we consider in this paper, we can only do our best to design the algorithm and the representation of the problem so that it finds an as good as possible policy. While we can not expect to prove anything about the quality of the policy that will be learned, it may be assessed experimentally.

Learning starts with an arbitrary policy, and the algorithm interacts with the problem to be solved by performing actions according to the states being visited and its current policy. Gradually, the RL agent adapts its policy to optimize the objective function, i.e. to minimize the makespan in this paper. Learning is done using the algorithm described in Section IV-A.

IV. Algorithm

A. Actor-Critic

Among the various RL algorithms that may cope with the problem at hand, we choose an "actor-critic" algorithm, known as A2C [41], a very popular RL algorithm. The policy is represented by a neural network. The neural network takes a state as input and produces a probability distribution on the set of possible actions as output, which should reflect their expected performance when performed in the input state. Since it is impossible to exhaustively visit all states within a reasonable amount of time, we take advantage of the ability of a neural network to generalize from its observations. The RL agent learns to maximize the objective function $R(\text{makespan})$. To do so, it modifies the current policy to obtain a better one. At any moment, the agent follows its current policy π. To assess its quality, the RL agent performs steps until a terminal state is reached: at each step t, the quadruple (s_t, a_t, r_t, s_{t+1}) is collected. After a batch of such quadruples has been collected, the weights of the network are corrected in order that the poicy is improved. This correction is performed with a stochastic gradient descent, as customary in neural

networks. This improvement step is repeated many times until a local optimum is reached (local optimality is the only guarantee we can expect.)

Though it is impossible to fully describe how an actor-critic algorithm works within the required length of this paper, we provide a few details about A2C. A2C uses two neural networks: i) a policy network (the "actor") $\pi_\theta(a_t \mid s_t)$ with weights θ which computes a distribution of probability over the actions; ii) a value network (the "critic") with weights θ_v which estimates the value of a state $V_{\theta_v}(s_t)$. To simplify notations, we drop the θ subscripts in the following. Both networks have to be optimized so that the resulting policy performed by the agent optimizes the objective function $R(\text{makespan})$. For that purpose, each network has its own objective function to optimize. The value of a state $V(s)$ quantifies how good it is to be in a state s in order to optimize the objective function: this value is not known: it is learned along with the policy. The estimation of V is improved by minimizing the mean square error of Bellman's function.

The optimization of π relies on the policy gradient theorem that relates the gradient of the objective function and a certain function of π, namely: $\nabla_\theta \log \pi(a \mid s) A(s,a)$. Here, $A(s,a)$ denotes the advantage function which measures how action a is better than the average in state s. A is estimated empirically with the quadruples (s_t, a_t, r_t, s_{t+1}) and the current estimate of V. We also add the entropy of the policy to the objective function minimized by the policy network, yet an other trick known to improve exploration [49]. We end-up with the problem of minimizing $\nabla_\theta \log \pi(a \mid s) A(s,a) + \beta \nabla_\theta \mathcal{H}(\pi(s))$, where $\mathcal{H}$ denotes the entropy function of a probability distribution and β a hyper-parameter which controls the influence of the entropy regularization.

B. Architecture of the RL agent

In this paragraph, we very briefly outline the architecture of READYS. The neural network architecture is kept as simple as possible in order to minimize the scheduling computation overhead (see Fig. 2). The input of the neural network(s) is made of the DAG information and the state of the computing resources. As explained earlier, a stack of graph convolution layers mixes the information of the nodes of the DAG, up to a certain depth from the current node. The output of the last convolution layer is used as an internal representation of the DAG. The number of GCN layers is a parameter of the algorithm; it should be related to the size of the window w as at least w layers are needed to allow information to flow between the nodes of the sub-DAG to the available tasks. Empirically, we found that using w layers is enough. Between these layers, we use ReLU functions as non-linear activations. This representation of the DAG is stacked together with the embedding of the computing resources described in III-B and used to compute final action probabilities. We then sample an action according to this output probability, and schedule the corresponding task on the current processor (or do not schedule anything if it is the $\emptyset$ action). This architecture being quite complex and impossible to present with all the details needed to re-implement it, we provide our own implementation in [27].

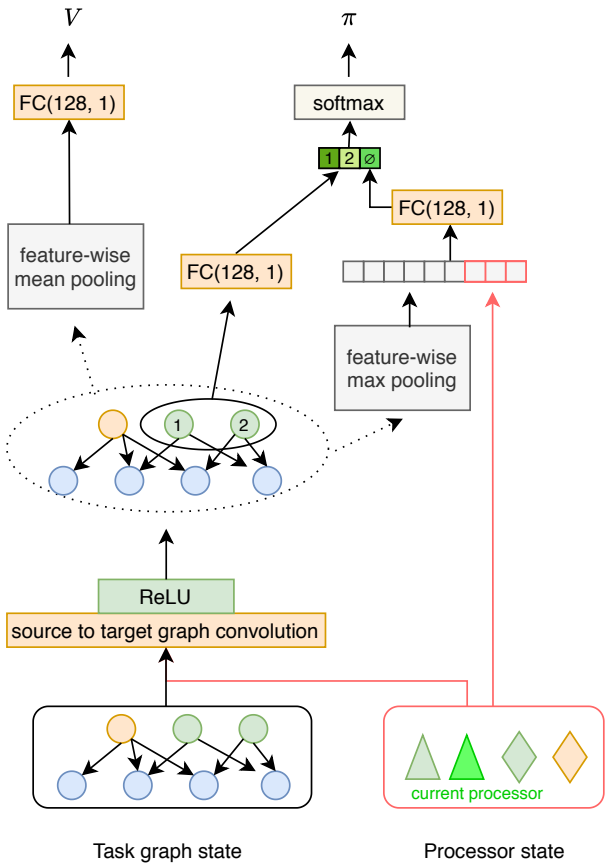

Figure 2. Overview of the architecture of READYS, the RL agent. At the bottom, a sub-DAG enriched with the computing resource state information is fed into a stack of several graph convolution layers and output an internal representation. It is used to estimate the state value V via mean-pooling and one-dimensional projection. The embeddings of available tasks (here 1 and 2 are aggregated into a batch and projected onto a one-dimensional vector, which can be seen as the score of each task. This vector is concatenated with a single real number, the score of the $\emptyset$ action, computed from a projection of the processor state and the max-pooling of the internal DAG representation, and normalized with a softmax to output probabilities π. FC(128, 1) denotes a fully connected layer with an input size of 128 and an output size of 1. We represent each type of processor (*eg* CPU and GPU) by a different shape. Idle processors are in green, running processors are in orange, and the current processor is in light green.

V. Experiments

In this section, we report experiments in which we compare READYS with other well-known and efficient heuristics.

A. Task Graphs

We consider three types of DAGs corresponding to Cholesky [6], LU [3], and QR factorizations [4]. These applications are used in many real-life applications and are considered as a good testbed for the evaluation of runtime systems [18], [19]. For instance in [10] the authors use a Cholesky factorization to solve an electromagnetic problem ; in [14] authors use an dense QR factorization for solving an

eigenvalue problem that can be applied in quantum chemistry, finite element modeling or multi-variate statistics. From a computer-science point of view, These factorizations involve (i) a large number of tasks, (ii) complex dependencies and (iii) a small number (4) of different kernels. Therefore they constitute a very good benchmark for scheduling algorithms [5], [29] and designing good scheduling policies in this context is both very meaningful theoretically and of extreme practical importance.

B. Simulation Model

For a task-processor pair (i, p), we denote by $E(i, p)$ the expected duration of task i executed on p, and $d(i, p)$ its actual duration. d is a stochastic variable. In the experiments, d is obtained by adding a Gaussian noise to the expected duration E. More formally, we model d as follows:

$$d(i,p) = \max\left[0, \mathcal{N}\Big(E\big(i,p\big), \sigma E\big(i,p\big)\Big)\right],$$

where σ is a parameter of the simulated environment controlling the noise level: the greater σ, the larger the uncertainty on the duration of each task. We are aware of the limits and drawbacks of this duration model. There does not exist any good model in the literature that would fit the setting under study, but the model proposed here enables us to model some uncertainty on task duration, which is an essential feature of the systems we are considering. We leave to future work either to come up with a good model, or to study the sensitivity of our analysis to various noise models. In our experiments, the expected durations of each kernel of each type of graph for each type of resource (CPU and GPU) are taken from real measurements of the literature [3], [4], [6].

C. Baselines

Our goal is to analyze the performance of a reinforcement learning based algorithm for the dynamic scheduling of Cholesky, LU and QR factorizations. In practice, it is very difficult to accurately predict the computational costs and the communication durations in an HPC environment in which the various running processes unpredictably influence the execution times of each others. This explains the success of dynamic schedulers [8], [16], [22]. Indeed, since it is not possible to schedule and allocate tasks long in advance, in practice, dynamic runtimes rely solely on the description of the machine state and on the tasks already performed, using a task priority mechanism to define which tasks to perform in the event that the number of available resources is less than the number of available tasks. In these dynamic systems, task placement decisions are made a little in advance, taking into account the placement of input data, and this delay is used to transfer task input data if necessary to overlap communication and computation. We have already discussed in Section III-A how to overlap communications and computations in both the CPU multicore and the GPU cases.

We have chosen HEFT [48] as a reference static algorithm. It is a *static* list-scheduling heuristic that contrary to READYS uses the whole DAG to compute a schedule. This consists in never leaving a resource inactive if there exists a ready to be processed task and breaking ties among candidate tasks by choosing the one that is farthest from the end of the computation. In the homogeneous case, it corresponds to the one with the longest critical path. It has been demonstrated in [12] that despite its simplicity, this strategy gives excellent results for CHOLESKY factorization, especially when execution times are similar to what is observed in practice on GPUs. We also compare READYS to MCT (minimum completion time). MCT is a *dynamic* heuristic that, similarly to our approach, considers tasks one after the other without considering the whole DAG. Each time a task becomes ready it is assigned to the resource where it is expected to complete the soonest [46].

D. Training

We perform a random search on several hyper-parameters of our model: the window $w \in [0, 2]$, and the number $g \in [1, 3]$ of GCN layers. The networks are trained using the Adam optimizer with a learning rate of 0.01, while leaving the over hyper-parameters default in PyTorch [44].

Regarding the actor-critic algorithm, we chose a discount factor $\gamma = 0.99$, a baseline loss scaling of 0.5, and grid-searched the unroll length in $[20, 40, 60, 80]$ and the entropy loss ratio in $[10^{-3}, 5 \times 10^{-3}, 10^{-2}]$. Informally, we noted that training an agent would take approximately 20 minutes on a standard laptop with no GPU.

E. Results

The performance of several models trained on the three types of DAGs, for different number of tiles, CPUs and GPUs are summarized in Figure 3. We recall that T is the number of tiles in each dimension of the matrix, hence T^2 tiles in total, and that there are $O(T^3)$ tasks in the considered DAGs [3], [4], [6]. We compute the improvement over HEFT (static heuristic) and MCT (dynamic heuristic). As soon as $\sigma > 0$, durations are stochastic and reported figures are obtained by averaging the performance over 5 runs/seeds. We see that when σ is small, READYS performs similarly to HEFT (red boxes). One must keep in mind that HEFT makes use of the complete DAG to compute a schedule whereas READYS does not as it is fully dynamic and discovers the graph on-line. As soon as σ increases, READYS outperforms HEFT taking advantage of the fact that it discovers task durations by itself. Compared to MCT (blue boxes) which is a fully dynamic heuristic like READYS, we see that our approach is much more efficient even for low noise, exhibiting more than 35% of improvement in some cases. Moreover, the relative performance of MCT compared to READYS is roughly constant when σ varies when the task graph is sufficiently large ($T = 6$ or more). Indeed, they are both insensitive to uncertainty about the duration of tasks (unlike HEFT), because they schedule tasks taking into account the actual state of the system and the unexpected duration of the tasks. On the opposite, HEFT computes the schedule before the execution and is less able to cope with duration variability. It also means that in order to deal with uncertainty, as soon as the input graph is large enough, it is

Figure 3. Makespan improvement over HEFT and MCT according to $T \in \{2, 4, 8\}$ (rows), the noise level σ, and for each of the 3 tasks we consider (3 columns), when the computing platform is made of 2 CPUs and 2 GPUs. The larger the bars above 1, the better READYS performs w.r.t. to competitors.

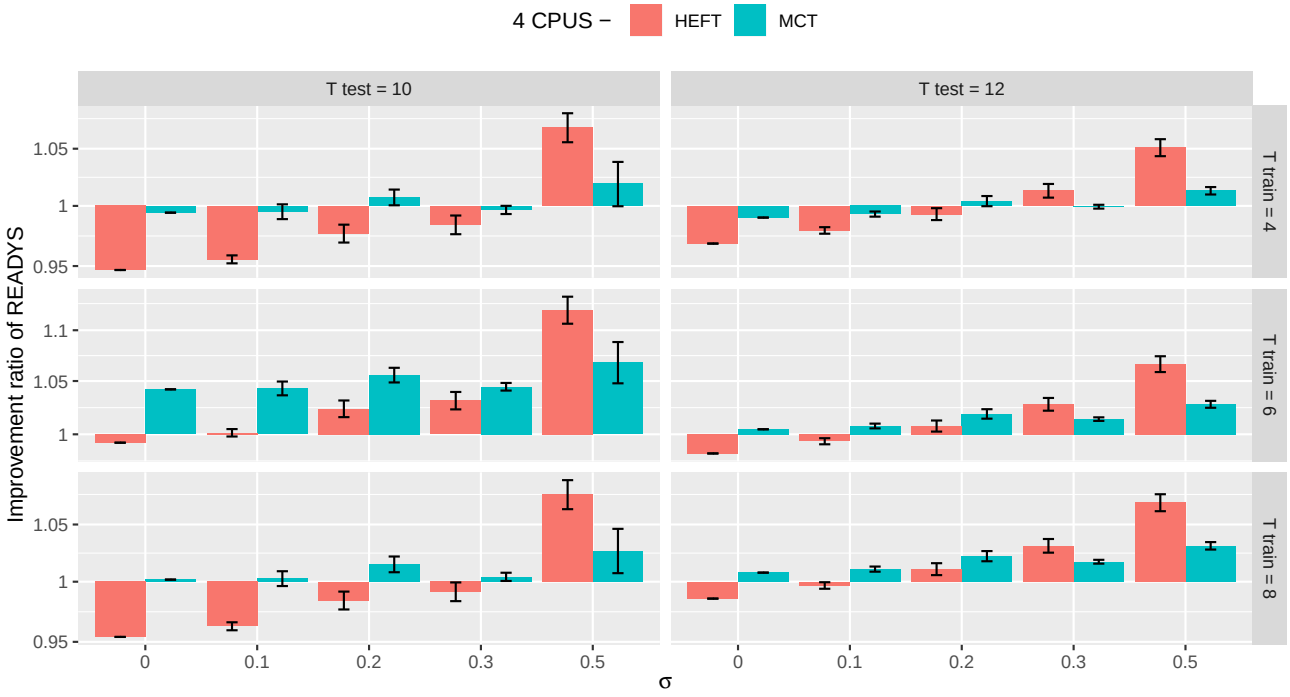

Figure 4. Transfer learning experiments: Makespan improvement over HEFT for the CHOLESKY task graph for several noise levels σ. On the left, the testing DAG is the tiled CHOLESKY with $T = 10$ tiles, and on the right the tiled CHOLESKY with $T = 12$ tiles. The computing platform is made of 4 CPUs.

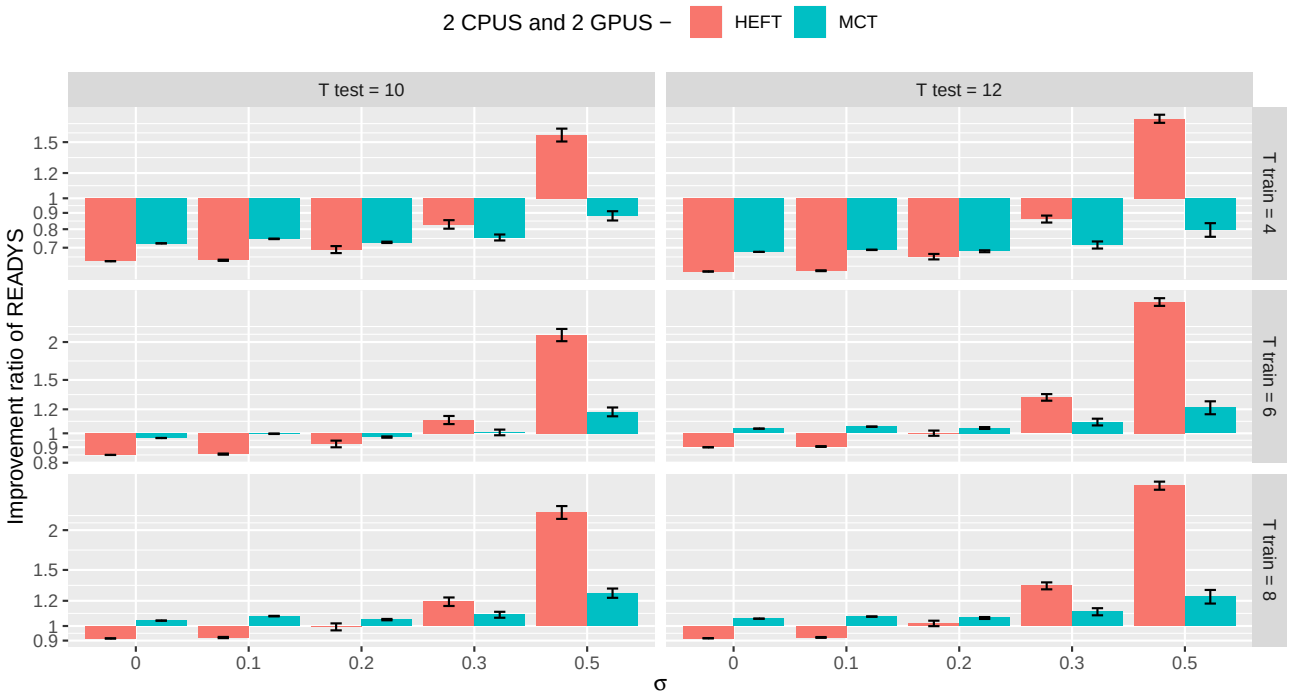

Figure 5. Transfer learning experiments: Makespan improvement over HEFT for the CHOLESKY task graph for several noise levels σ. On the left, the testing DAG is the tiled CHOLESKY with $T = 10$ tiles, and on the right the tiled CHOLESKY with $T = 12$ tiles. The computing platform is made of 2 CPUs and 2 GPUs.

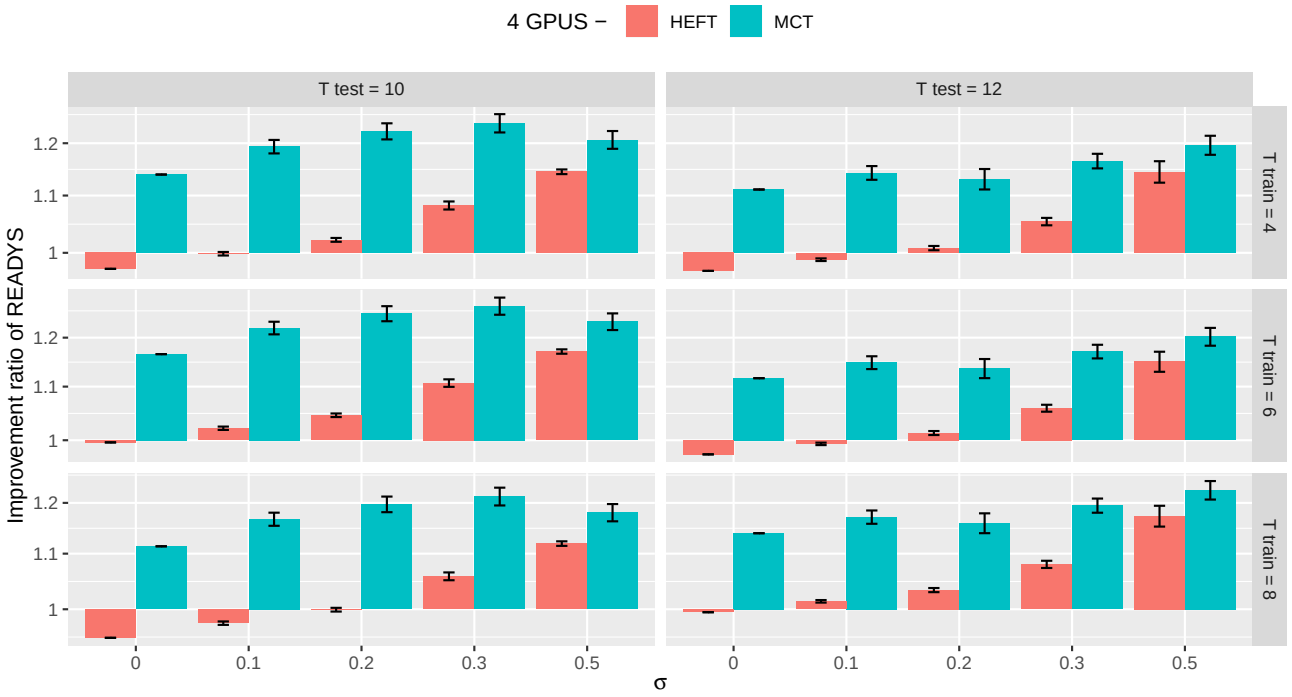

Figure 6. Transfer learning experiments: Makespan improvement over HEFT for the CHOLESKY task graph for several noise levels σ. On the left, the testing DAG is the tiled CHOLESKY with $T = 10$ tiles, and on the right the tiled CHOLESKY with $T = 12$ tiles. The computing platform is made of 4 GPUs.

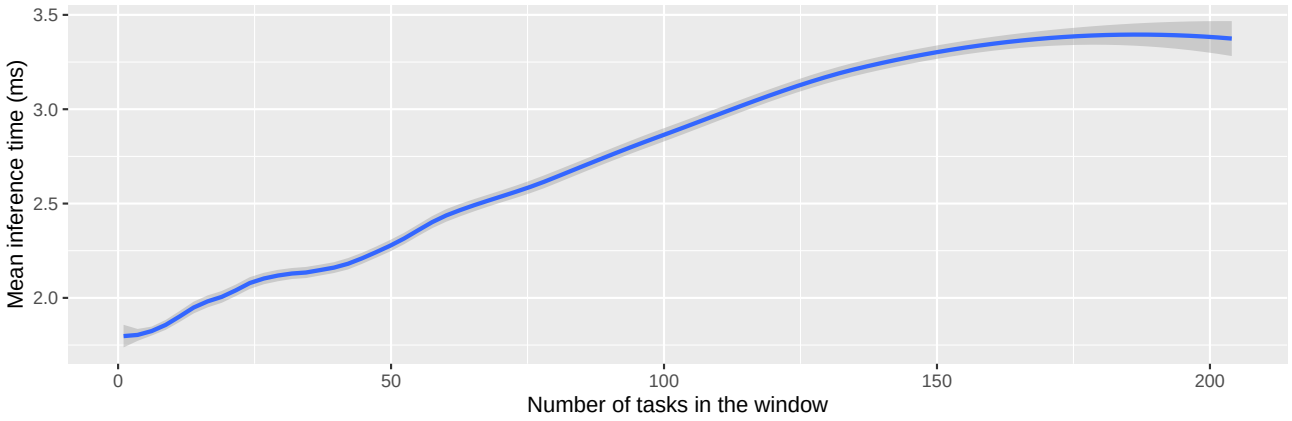

Figure 7. Mean inference time for the CHOLESKY DAG with 99% confidence interval.

better to make dynamic decisions than to have a complete view of the topology of the graph.

F. Transfer learning

Training an RL agent is well-known to be time consuming. To overcome this difficulty, we investigate whether an agent trained to schedule a specific DAG is able to schedule other DAGs of different sizes, which is an example of transfer learning. Reducing learning time is crucial to make reinforcement learning usable in practice. In particular, it is crucial that a scheduling policy learned on a small size graph may be transferred to a larger size graph, that would require too much learning time to train from scratch.

We consider the CHOLESKY task graph and apply directly READYS trained on either $T = 4, 6$ or 8 tiles (that is respectively 20, 56 and 120 tasks) to DAGs of size 10 and 12 tiles (220 and 364 tasks). Results are summarized in Figure 4, Figure 5 and Figure 6. As previously, figures are averaged over 5 runs/seeds when stochastic. These experiments exhibit very promising transfer capacities for all three considered computing platform architectures. Models trained for $T = 6$ and $T = 8$ obtain roughly similar performances when used to schedule problems of size 10 or 12, losing by only a few percents against HEFT when $\sigma = 0$, and becoming again competitive as soon as $\sigma > 0.2$. The results are even better when compared to MCT where the improvement is always positive.

We can notice that the performance of READYS varies according to computing platforms, which is not surprising as the optimal scheduling strategy may change a lot. In the case of 4 GPUs for example, scheduling tasks on the critical path is crucial, which can be difficult for baselines like MCT, hence the large improvements of READYS.

As expected, models trained for $T = 4$ obtain weaker performances: the environment used for their training is too different from the testing environment. For instance, the ratio between the different types of kernels is too different in this case. When dealing with large task graphs, this may suggest an alternative strategy to a costly training from scratch: finding an intermediate instance close enough to the initial DAG, which size is small enough so that training time remains short; train on it, and apply the learned policy to the larger instance.

G. Inference Time

We further report wall-clock inference time on standard hardware (one CPU, no GPU) in Fig. 7 in order to evaluate the scheduling overhead, since scheduling decisions are made at runtime. It increases with the number of tasks in the window (in our experiments, the average number of tasks in the window is 45), but remains in the order of the milliseconds, so that the scheduling overhead is reasonable as in the case of tiled algorithm kernels execution time is much larger.

VI. CONCLUSION AND FUTURE WORK

In this paper we address the following question: *can Reinforcement Learning effectively be used to dynamically schedule Directed Acyclic Graphs (DAGs) on heterogeneous systems?* This is both a very difficult question as scheduling is a NP-Hard combinatorial optimization problem and a very important question as dynamic scheduling is used in many task-based runtime systems. To the best of our knowledge, we are the first to positively answer this open question.

To do so, we consider several DAGs arising from linear algebra. We demonstrate the ability of READYS to be competitive with state-of-the-art static scheduling algorithms such as HEFT even when there is no noise in the task duration estimation. This is remarkable as HEFT is a static heuristic that contrary to our approach, uses a full knowledge of the graph (topology and tasks durations). As READYS makes very little use of prior knowledge about the environment, it is particularly powerful when the uncertainty about the task duration is large or when the environment is stochastic, improving the results obtained by HEFT by a large margin. Compared to a dynamic approach such as MCT, the results are even better as we are able to outperform MCT in all cases, regardless of the uncertainty of task durations.

Learning scheduling algorithms for parallel heterogeneous computing platforms capable of handling stochastic duration is

a key feature of our solution, since real execution environments do not generally behave in a deterministic way (e.g. regarding resource availability, the execution time of a given task, the communication time of a given transfer). In this case, reinforcement learning is capable of adapting to current execution conditions, dealing with unplanned situations. Moreover, and very importantly, we show that our proposed solution enables *transfer learning*. A model trained on a specific DAG of a (small) given size is able to efficiently apply the learned strategy to larger graphs.

This work opens several directions for future works. We use A2C as our reinforcement learning algorithm. Other algorithms that have been recently introduced (*e.g.* [23]) may improve our results still further. Future work could include generalizations of transfer performances, using for example techniques from few-shot learning or meta-learning. This ability to generalize and transfer knowledge is crucial: paying the full price of model training is probably the main practical obstacle to using these techniques. More broadly, this paper opens new avenues for the use of reinforcement learning for scalable and practical dynamic DAG scheduling.

ACKNOWLEDGEMENTS

Experiments presented in this paper were partially carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations. NG is recipient of a PhD funding from AMX program, Ecole polytechnique.

REFERENCES

[1] Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving np-hard problems on graphs by reinforcement learning without domain knowledge. *CoRR*, abs/1905.11623, 2019.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv:1906.08879 preprint*, 2019.

[3] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU factorization for accelerator-based systems. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 217–224. IEEE, 2011.

[4] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. QR factorization on a multicore node enhanced with multiple gpu accelerators. In *IPDPS'11*. IEEE, 2011.

[5] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, Samuel Thibault, and Stanimire Tomov. Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Knoxville, United States, July 2010.

[6] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. Are static schedules so bad? a case study on cholesky factorization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1021–1030. IEEE, 2016.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[8] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[9] M.Emin Aydin and Ercan ztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2):169–178, 2000.

[10] Marc Baboulin, Luc Giraud, and Serge Gratton. A parallel distributed solver for large dense symmetric systems: applications to geodesy and electromagnetism problems. *The International Journal of High Performance Computing Applications*, 19(4):353–363, 2005.

[11] Olivier Beaumont, Lionel Eyraud-Dubois, and Yihong Gao. Influence of tasks duration variability on task-based runtime schedulers. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 16–25. IEEE, 2019.

[12] Olivier Beaumont, Julien Langou, Willy Quach, and Alena Shilova. A makespan lower bound for the scheduling of the tiled cholesky factorization based on alap scheduling. In *Proceedings of the 26th International European Conference on Parallel and Distributed Computing (EuroPar 2020)*, pages 1–12, 2020.

[13] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *CoRR*, abs/1811.06128, 2018.

[14] Paolo Bientinesi, Francisco D Igual, Daniel Kressner, and Enrique S Quintana-Ortí. Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. In *International Conference on Parallel Processing and Applied Mathematics*, pages 387–395. Springer, 2009.

[15] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38((1-2)):37–51, 2012.

[16] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[17] Peter Brucker and Sigrid Knust. Complexity results for scheduling problems, 2020. http://www.informatik.uni-osnabrueck.de/knust/class/.

[18] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[19] Jaeyoung Choi, Jack J Dongarra, L Susan Ostrouchov, Antoine P Petitet, David W Walker, and R Clint Whaley. Design and implementation of the scalapack lu, qr, and cholesky factorization routines. *Scientific Programming*, 5(3):173–184, 1996.

[20] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier. Resource aggregation for task-based cholesky factorization on top of modern architectures. *Parallel Computing*, 83:73–92, 2019.

[21] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS17, page 63516361, Red Hook, NY, USA, 2017. Curran Associates Inc.

[22] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02):173–193, 2011.

[23] Yannis Flet-Berliac, Reda Ouhamma, Odalric-Ambrym Maillard, and Philippe Preux. Learning Value Functions in Deep Policy Gradients using Residual Variance. In *Proc. ICLR*, 2021.

[24] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*, pages 1676–1684, 2018.

[25] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[26] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of discrete mathematics*, volume 5, pages 287–326. Elsevier, 1979.

[27] Nathan Grinsztajn. *RL for Dynamic Scheduling https://github.com/nathangrinsztajn/RL_for_dynamic_scheduling*, 2021, [Online].

[28] Nathan Grinsztajn, Olivier Beaumont, Emmanuel Jeannot, and Philippe Preux. Geometric deep reinforcement learning for dynamic DAG scheduling. In *Proc. ADPRL*. IEEE Press, 2020.

[29] Emmanuel Jeannot. Symbolic mapping and allocation for the cholesky factorization on numa machines: Results and optimizations. *The International journal of high performance computing applications*, 27(3):283–290, 2013.

[30] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[31] Vaibhav Kumar, Siddhant Bhambri, and Prashant Giridhar Shambharkar. Multiple resource management and burst time prediction using deep reinforcement learning. In *Eighth International Conference on Advances in Computing, Communication and Information Technology*, pages 51–58, 2019.

[32] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1-3):259–271, 1990.

[33] Joseph YT Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC press, 2004.

[34] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *CoRR*, abs/1810.10659, 2018.

[35] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[36] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks - HotNets '16*, pages 50–56. ACM Press, 2016.

[37] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270288, New York, NY, USA, 2019. ACM.

[38] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning*, Proceedings of Machine Learning Research, pages 2430–2439, 2017.

[39] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proc. ICML*, pages 2430–2439, 2017.

[40] Akash Mittal, Anuj Dhawan, Sahil Manchanda, Sourav Medya, Sayan Ranu, and Ambuj K. Singh. Learning heuristics over large graphs via deep reinforcement learning. *CoRR*, abs/1903.03332, 2019.

[41] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[42] Alexandru Iulian Orhean, Florin Pop, and Ioan Raicu. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 117:292–302, 2018.

[43] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. In *Proc. ICLR*, page 24, 2020.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[45] E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3), 2009.

[46] Rizos Sakellariou and Henan Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 111. IEEE, 2004.

[47] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[48] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[49] Ronald J. Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991.

[50] Qing Wu, Zhiwei Wu, Yuehui Zhuang, and Yuxia Cheng. Adaptive DAG tasks scheduling with deep reinforcement learning. In Jaideep Vaidya and Jin Li, editors, *Algorithms and Architectures for Parallel Processing*, volume 11335, pages 477–490. Springer International Publishing, 2018.

[51] W. Zhang and T.G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proc. IJCAI*, pages 1114–1120, 1995.

[52] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint arXiv:1910.01578*, 2019.