Scheduling with Precedence Constraints in High-Performance Computing

Second Year PhD Continuation Report

Thomas McSweeney[1]

19th August 2019

[1]School of Mathematics, University of Manchester, Manchester, M13 9PL, England (thomas.mcsweeney@postgrad.manchester.ac.uk).

Table of contents

Chapter 1

Introduction

The most powerful modern *high-performance computing* (HPC) machines are fast approaching *exascale*, capable of performing at least 10^{18} floating-point operations per second. This represents orders of magnitude more raw computational power than was available even five years ago. Similarly, the increasing popularity of systems with *heterogeneous* processors offers us the potential for almost perfect efficiency by ensuring that all of our computations are performed on the resources that are best suited to handle them. In addition, the corresponding growth in the total number of processing cores available gives us more potential for parallel computations than ever before. But it has become apparent that these prevailing hardware trends will also necessitate changes to our traditional programming paradigms in order to fully exploit the awesome potential of this changing landscape.

This research is centered around a problem which can perhaps be most concisely summarized by the following question: *how do we most efficiently execute large computational jobs on the diverse processing resources of modern HPC systems?*

Since modern supercomputers often have annual energy costs in the millions of dollars, efficiency is even more important than ever before so this is where the problem is most acute. But ultimately it is relevant in more general computing contexts. Indeed, we expect this to increasingly be so over the next few years as the current hardware trends in HPC inevitably filter downward to everyday computing environments.

The remainder of this chapter is structured as follows. In Section 1.1, we briefly sketch the current HPC landscape and the most pertinent trends in recent years. In Section 1.2 we introduce *task-based parallelism*, a programming paradigm designed to harness these changes, and then more precisely state the problem which this research seeks to address in that context. In Section 1.3 we discuss some of the relevant existing software. Finally in Section 1.4, we conclude by explicitly defining the scope of this research and the assumptions we will make going forward.

1.1 Trends in high-performance computing

About fifteen years ago, it became apparent that although Moore's Law continued to hold true, the performance improvement of single-core processors had begun to stagnate [12]. This was primarily because physical upper limits on their clock frequencies had been reached due to heat, power and current leakage issues, a problem sometimes referred to as the *frequency wall* [4, 80]. To overcome this hurdle, designers instead began to increase the number of processing cores on each individual chip.

Thus we entered the multicore[1] era. Today, even mainstream computers—desktops, laptops, tablets, games consoles—now almost universally have multicore processors and modern HPC systems themselves may have hundreds of thousands or even millions of cores. The Oak Ridge National Laboratory's Summit machine, the world's fastest supercomputer as of this writing, has a total of $2,397,824$ cores [87].

Although the move towards multicore allowed processors to maintain constant performance improvement, energy issues soon came to the forefront. As processors have grown more and more powerful, their energy consumption and heat production has grown in kind; this problem is often called the *energy wall* [23]. Computing systems with many different kinds of processors have thus became increasingly attractive. The idea is that by performing vital (or compute-intensive) tasks on higher power processing units and less pressing (or compute-intensive) tasks on lower power ones, we can reduce wasted computational effort and therefore improve overall energy efficiency [23, 32]. For this reason, modern HPC architectures are increasingly likely to be heterogeneous in this respect; this is true for more than a fifth of the machines on the most recent TOP500 list of the world's most powerful supercomputers [87]. Among the Green500, the list of those machines that provide the best performance (in terms of *floating point operations per second* or *flops*) per watt of energy supplied, all of the current top ten are heterogeneous [31]. It is expected that this trend will only accelerate in the future [5, 12].

Heterogeneous HPC architectures today often consist of just two kinds of processors: CPUs and some more powerful alternative processor, usually called *accelerators*. We shall refer to such systems as *accelerated* throughout this report. Initially research was typically focused on how these accelerators could be used as a replacement for the CPUs or how work could be offloaded to them with little overall coordination. However, the consensus is now shifting toward the idea that to achieve the best possible performance we need to design systems that seamlessly integrate the different types of processors [56].

[1] A distinction is sometimes made between *multicore* and *manycore*, with the latter generally being used for specialized processors which are explicitly designed to exploit a high degree of parallelism and which generally have upwards of dozens of cores; the difference is not important here however so we will exclusively use the former term.

A wide variety of devices have been employed in this fashion in HPC systems. For example, *field-programmable gate arrays* (FPGAs), integrated chips that can be programmed for specialized tasks, are used by Microsoft's Project Catapult in network servers in the company's data centers [55]. Other machines use powerful but more traditional manycore processors such as the 2048-core PEZY-SC2 [101].

However the real driver of this trend in the last few years has been the *graphics processing unit* (GPU[2]). Fueled by their importance in the gaming industry, GPUs have become increasingly cheap and powerful in recent years and, being optimized for graphics rendering applications, have proven to be adept at many other large, parallel tasks which access memory in a regular manner [66]. For example, in numerical linear algebra, GPUs have often been found to deliver superior performance to CPUs when multiplying or factorizing very large matrices, a classic *embarrassingly parallel* operation, but the converse may be true when performing smaller, more serial tasks [1, 105].

In any multiprocessor computing environment how those processors communicate with each other is clearly of the utmost importance. Since this communication is usually data movement, this means how memory is distributed between them. In the past, it was not usually impossible for CPUs to share memory with alternatives such as GPUs, although more recent designs attempt to overcome this. However, this is still true to an extent and one popular approach, at least within nodes, is for CPUs to share common memory but to require an *interconnect* to send messages (e.g., data) to the other kind of processors.

Of course, although energy concerns may have motivated the move towards heterogeneous architectures, it also simply makes sense on a more fundamental level. Different kinds of processors have different attributes, such as their power or how they access memory, and therefore different sorts of computational tasks that they are good at. So if we want to achieve the best possible overall performance, why don't we ensure that we perform all of the constituent parts of an application on the kind of processor best suited to handle it?

It has not escaped notice that this guiding principle of heterogeneous computing applies even to cores on the same chip. Processors of this kind are becoming increasingly common and offer promising performance gains, particularly with regard to energy efficiency [44, 72, 96]. Major manufacturers such as Intel and AMD have begun to produce heterogeneous chips which integrate CPUs and GPUs onto the same piece of silicon for the consumer market and some expect this to become the default in the near future [97].

Although high-performance computers have been almost universally parallel since at least the 1990s, the massive growth in the number of processing cores available in the multicore era offers much more scope for parallel computation than ever before. Increasingly, machines have a hierarchy of potential paral-

[2]Some prefer to use GPGPU (for *general purpose graphics processing unit*) when they are used for general computations but we will stick with the shorter acronym.

lelism: across different nodes, between the processing units within those nodes themselves, among the cores of *those* processors, and so on. This means that now almost all codes we intend to run on these machines must be parallelizable in order to take full of advantage of the available resources. This is problematic only in light of the fact that parallel computing in general has traditionally proven difficult from a programming perspective [26, 61] and this is compounded by the complex hierarchical parallelism of modern HPC systems [98]. From a programming perspective, the question therefore is, how do we successfully exploit all of this new potential parallelism without departing too radically from our current programming practices? After all, we want retain existing programs and software that have proven successful as much as possible.

HPC architectures now almost exclusively comprise many separate nodes connected in a network working together so that the user can essentially consider them to form a single system. Note that in this context a node is any discrete computing unit, which may be a physical hardware device but could even be a software abstraction (such as a thread). Depending on the system, the complexity of the nodes themselves can range from simple single processor devices to manycore machines which may themselves be distributed systems.

In order to ensure that the constituent nodes of a distributed system work together effectively, they need to communicate efficiently. Details of how this is generally done are mostly outside our scope, but an accessible recent overview can be found, for example, here [94]. For our purposes, it suffices to say that this is a mature topic that has been studied in depth and that the prevailing modern approach is *message passing*, in particular the *Message Passing Interface* (MPI) standard [6].

Distributed HPC systems can generally be divided into two types: *cluster* or *grid* computers. The former refers to systems comprising multiple workstations connected in a local network, with each node generally being quite similar and running the same operating system (i.e., usually homogeneous). By contrast, grid computing refers to systems comprised of multiple nodes that may be very different in terms of both hardware and software, which may not even be administrated centrally [94]. *Cloud computing* can be viewed as an extension of grid computing: essentially, users can construct the infrastructure they want from a large pool of available resources, usually over the internet.

The enormous number of cores available in modern HPC systems have proven difficult to incorporate into strictly shared memory architectures due to power issues so memory is usually distributed across the system in some way. For example, one popular approach is for memory to be shared within nodes but distributed across the system itself, with an interconnect used for communication between the nodes. Specialized interconnects are used in modern HPC systems but communication between processors on different nodes is still often expensive. Of course this is an area where performance continues to improve but in general this improvement isn't as rapid as the corresponding increase in computational

power of the processors themselves. The rule of thumb in HPC remains that *computation is cheap but communication is expensive*.

1.1.1 Summary

To briefly summarize, modern HPC systems may:

1. Possess many heterogeneous processing resources.
2. Have a hierarchy of parallelism.
3. Have a complex memory architecture.

1.2 Task-based programming

One paradigm that has become popular in recent years for harnessing the changes in HPC hardware is *task-based parallel programming*. The basic idea is that we divide the application we wish to execute into a collection of *tasks*—logically discrete atomic units of work—and then identify which of these can be done in parallel and which cannot by specifying all of the *precedence constraints* or *dependencies* between them. These are usually of the form "Task A needs some data from Task B before it can be executed".

One of the major advantages of task-based programming is that writing parallel programs becomes much easier. The programmer basically just needs to implement sequential code to be executed at the task level and specify the dependencies between the tasks (usually data movement). Likewise, task-based programming achieves the aim of allowing us to retain efficient existing serial code since this can still be used at the task level.

Another benefit of task-based programming is that we can equate all applications with a *task dependency graph*, where each node of the graph represents a task and edges the dependencies between them. This makes the code extremely portable. In this report we are interested only in applications whose task graphs are *directed acyclic graphs* (DAGs)—directed and without any cycles. Many scientific computing applications can be expressed in this form.

Another reason why the abstraction from applications to DAGs is so useful is that it allows us to bring techniques from other areas of mathematics to bear; we know quite a lot about graphs. In particular, the concept of the *critical path* is important. The name itself comes from project management, where it is the longest sequence of activities that must be done in order to complete a project [42]. For a graph, the critical path is the longest of all paths through it. This is useful because, if we assume sufficient parallelism, then the time it takes to execute the tasks on the critical path of a DAG therefore gives a lower bound on the total execution time of the entire DAG itself. We will see in Chapter 2

that the idea of planning along the critical path of a DAG underlies many of the algorithms that we consider in this report.

We can distinguish two classes of task-based programming models: *sequential task flow* (STF) models and *parameterized task graph* (PTG) models. The key difference between the two is in how they represent the DAG from the application code. STF models construct the entire DAG based on the order in which tasks are inserted and how data is accessed, whereas PTG models express tasks and their dependencies symbolically and extract the portion of the DAG relevant to the tasks at hand, so in particular never see the entire DAG at once [111]. Examples of software using both models are discussed in Section 1.3.

One application area in which task-based programming has proven to be particularly popular is *numerical linear algebra* (NLA). This is largely because of the structure of linear algebra algorithms themselves and also because task-based programming allows us to make use of efficient existing BLAS implementations [47]. Typically, algorithms operating on large matrices proceed by dividing the matrix into smaller *tiles* (or *blocks*) and using BLAS routines on those tiles. An example of how one common NLA algorithm—Cholesky factorization—can be implemented in such a way is described in Section 3.2.1. Numerical linear algebra was in fact the context in which this research was initially conceived and thus will frequently be used as a source of examples throughout this report.

1.2.1 The task scheduling problem

The immediate question that arises in task-based programming is, how do we find the optimal way to assign the tasks to the available processing resources while still respecting the precedence constraints? In other words, what *schedule* should we follow? Finding good solutions to this problem on heterogeneous (and, more specifically, accelerated) computing platforms is the main goal of this research.

It should be clear that the task scheduling problem is simply an instance of a more general combinatorial optimization problem (e.g., replace *processor* with *server* and *task* with *customer*), in which we have a set of jobs with various processing times and a collection of machines with different processing rates. This is known as the *job shop scheduling problem*, and has been extensively studied for many years. Perhaps the most notable special case of job shop scheduling is the famous *traveling salesman problem*, where we have a single machine (the salesman) and multiple jobs (the cities he must visit).

Unfortunately, as a general rule scheduling problems are usually NP-hard, except for some special cases [29]. The task DAG scheduling problem in particular is no exception and has long been known to be NP-complete, even for the simpler case of entirely homogeneous processors [88]. Assuming of course that P $\neq$ NP, this generally means that in lieu of efficient algorithms that are guaranteed to converge to optimal solutions we must make do with heuristics that give us reasonably good solutions in reasonable time. Many different approaches have

been proposed and we describe several examples in detail in Chapter 2.

Typically the aim in HPC is to minimize application runtime so throughout this report we assume that DAG weights represent computation and communication times, and the optimal schedule is that with the smallest *makespan*, the total length of the schedule. But weights may in theory represent any other variable that we wish to optimize, such as energy consumption. However, we do not consider the simultaneous optimization of (or tradeoff between) two or more different variables, although this is a topic we may pursue in future work.

In practice, task scheduling is typically handled by a runtime system to further alleviate the burden on the programmer [12]; two such examples are discussed in the following section.

1.3 Software

Task parallelism is currently very much in vogue so there is a good deal of software compatible with the paradigm. Perhaps most notably, OpenMP has supported task-based programming since version 3.0 was released in 2008 [63].

Notable task-based scheduling runtime systems for heterogeneous architectures include the following:

- StarPU [5]. Originally intended for multicore environments but more recently extended to distributed memory, it is of particular interest to us as it allows users to implement their own custom task scheduling algorithms. Based on the STF model. (Note that the name is not an acronym but should be read as **PU*. Linux users should hopefully then see why this is appropriate.)

- PaRSEC (*Parallel Runtime Scheduling and Execution Control*) [12]. Supports the scheduling of tasks in both shared and distributed memory accelerated systems. Uses the PTG model.

As already noted, task-based programming has proven particularly popular for numerical linear algebra. Examples of relevant software libraries are:

- PLASMA (*Parallel Linear Algebra for Multicore Architectures*) [2]. Comprising efficient implementations of parallel task-based versions of Level 3 BLAS routines, PLASMA only supports shared memory machines at present and is mainly intended for homogeneous systems. Originally built on a runtime system called QUARK but switched to OpenMP after the latter began to support task-based programming.

- DPLASMA (*Distributed Parallel Linear Algebra for Multicore Architectures*) [11]. An extension of PLASMA to distributed heterogeneous systems. Uses PaRSEC as its underlying runtime system.

- MAGMA (*Matrix Algebra for GPU and Multicore Architectures*) [85]. Limited to a single node with either CPUs and a GPU or multiple GPUs at present. Unlike PLASMA and DPLASMA, exclusively uses static task scheduling (see Chapter 2).

1.4 Scope of this research

While there is a substantial body of research concerning task scheduling on heterogeneous platforms, many of the existing algorithms and heuristics were designed for arbitrary platforms and thus may not be optimal for the particular case of accelerated systems comprising multicore CPUs and processors of only one other type, such as GPUs. Although by no means the only heterogeneous architecture around today, a significant number of HPC systems are of this form. Perhaps most notably, Summit, currently the world's fastest supercomputer, comprises $4,608$ nodes, each with two 22-core IBM Power9 CPUs and six NVIDIA Tesla V100 GPUs [45]. It is expected that such systems will become even more common in future, both in HPC and beyond. Some of the existing scheduling literature does address these platforms but it appears to us that there is still a clear need to consider this refinement of the task scheduling problem in greater detail. Hence for the remainder of this report, we restrict ourselves to computing environments comprising multicore CPUs and accelerators. Given the continuing phenomenal growth of GPU computing, we will generally assume that the accelerators are GPUs, but we expect that much of this research will be applicable for other accelerator types.

Of course, by focusing exclusively on accelerated platforms there is always the risk that this research may soon become obsolete because of the constantly evolving nature of HPC. Indeed, in the long-term it is expected that architectures will become much more heterogeneous than they generally are now, with multiple different processor types perhaps becoming the norm. This may well be the case but it is reasonable at to assume that accelerated systems will remain common into the near future at least. Indeed, as previously noted, we expect that everyday devices will become increasingly likely to be of this form in the next few years. The extension of this work to a wider range of heterogeneous architectures is something we intend to pursue in the future.

Another assumption that we make is motivated by the different manner in which we generally program CPUs and GPUs: we consider CPU cores to be individual processing units (PUs) but regard entire GPUs as discrete PUs. For example, we would regard a single node of Summit as comprising forty-four PUs of CPU-type and just six of GPU-type. This distinction in part motivates the assumptions we make about the memory distribution of the environments we consider. In particular, we assume that all CPUs share memory and thus any data movement costs are negligible, at least relative to the costs associated with

moving data between CPU and GPU (or two distinct GPUs). (Thus the target environments for this research are more similar to a single node of Summit than the entire machine itself.)

We make the following assumptions about the applications that we consider:

1. Processors can only execute a single task at a time.
2. All tasks are atomic and cannot be divided across multiple processing units.
3. All tasks can be executed on all processing resources, albeit with different processing times.
4. All tasks have only two possible computation costs, a CPU (core) execution time and a GPU execution time.
5. All computation and communication costs are fixed.
6. We have no input in how tasks are defined.

Note that the second assumption may not literally be true but is assumed to be so for the current level of scheduling. For example, it could be that a task consisting of multiple subtasks is scheduled on a single processor and then at a lower level those subtasks are allocated to the cores of the processor.

The third assumption means in particular that we do not consider constraints imposed by e.g., processor cache size, that may prevent certain tasks from being executed on some processors. As GPUs have grown more and more widely-used for general purpose computations, the range of tasks which they are incapable of performing at all has dwindled so this is not as restrictive as it once was. Of course, there are still some tasks that they cannot do and, conversely, tasks which must be done on GPU rather than CPU. However, we do not consider such applications here.

The fourth assumption reflects the fact that we are restricting this study to the case where we have only one type of CPU and GPU. Of course, in reality there is likely to be variation even between the execution time of a task on two different CPU cores but the motivation for this is that all of a task's CPU execution times (for example) are likely to be much closer to each other than they are to the GPU execution time(s).

The fifth assumption is arguably the most unrealistic. In practice, computation and communication costs will almost never be known precisely in advance due to the inherent difficulty of estimating, for example, the effective bandwidth available at execution time. The problem of scheduling task with precedence constraints under uncertainty is known as *stochastic* DAG scheduling, as these values are typically modeled as random variables from some (possibly unknown) distribution. While it has not been studied as extensively as the deterministic problem there is still a significant body of existing literature in this area

[16, 51, 82, 110]. The goal is typically to find methods that bound variation in the schedule makespan, which is notoriously difficult given that Hagstrom [34] showed that even computing its probability distribution is a $\#P$-complete problem[3].

Although we do not consider the stochastic problem in depth in this report, we do make occasional comments as our intention is to build on the work we have done here to address it in future research. In addition, we believe that reinforcement learning (see Chapter 5), one of the approaches to the problem that we will consider later in this report, may potentially be very powerful for many different aspects of the stochastic scheduling problem.

The last assumption can be more important than it may appear. In tiled numerical linear algebra algorithms for example, exactly how the matrix is tiled—i.e., the size of the tiles and thus the tasks in the DAG—is usually guided by the architecture on which the algorithm is to be executed; e.g., since GPUs are generally better suited to larger tasks than CPUs it makes little sense to divide the matrix into very small tiles on a multiple GPU platform. In future work we intend to consider the related problem of how to construct more amenable application task graphs for a given platform.

A fundamental distinction can be made between *static* and *dynamic* scheduling problems. Static schedules are fixed before execution and never subsequently altered: tasks are initially assigned to processors and never moved anywhere else. Dynamic scheduling algorithms use information gathered during execution to modify the initial schedule; a simple example would be moving a task scheduled to be executed later on a busy processor to one which is currently idle. There are generic advantages and disadvantages to both approaches. We need more information about the system for estimates (task execution times, etc) used in static schedules to be reliable, whereas dynamic schedules can compensate for any errors in their estimates during execution. But dynamic schedules can also be more expensive. For example, in order to move work from an overloaded processor to an idle one, we have the additional cost of moving data between the two.

It has been argued by some that static schedulers are no longer suitable because the complexity of modern heterogeneous architectures prevents accurate estimation before execution [5, 23]. A counterargument to this is that the ever-growing disparity between expensive communication and cheap computation may mean that dynamic schedulers are likely to be significantly more expensive than static ones on modern machines. Both of these arguments seem reasonable and the conclusion may be that we need to consider *hybrid* scheduling, which mixes the two. For example, we may have a system comprising many shared memory nodes for which communication costs between them are so high that we want to

[3]The traditional way to explain the complexity of this class intuitively is that it is equivalent to counting the number of solutions of an NP-complete problem [91].

statically schedule tasks to the nodes but dynamically schedule the tasks within those nodes themselves.

In this report we focus almost exclusively on static scheduling, but, as with the stochastic problem, we intend to consider dynamic DAG scheduling in future work and regard it as an area where reinforcement learning in particular may prove to be very useful.

Chapter 2

Existing static scheduling methods

Many different approaches for tackling the (static) task scheduling problem on heterogeneous platforms have been proposed over the last thirty years or so but they can basically be divided into two categories, *guided-random* and *heuristics*.

2.1 Guided-random

The basic idea behind these methods is that we start with a (possibly randomly-generated) set of schedules and then introduce some randomness to them in order to generate a new set of candidate schedules until we find one which is sufficiently good for our purposes [88]. Schedulers of this type often fit into the framework of more general optimization methods, such as *evolutionary algorithms*. These draw inspiration from biological processes and have the following basic structure:

1. Generate a random *population* of *individuals* (i.e., schedules).
2. Evaluate the *fitness* of each of them (i.e., how well the schedules satisfy the conditions we want).
3. Select the fittest individuals (however many we like) and *breed* them with each other—combine them in some biologically inspired way—to produce a new *generation* of individuals.
4. Repeat Steps 1–3 for the offspring, until a satisfactory individual is found.

Evolutionary algorithms have long been popular for difficult optimization problems such as scheduling and can often be the best way to find optimal (or nearly optimal) schedules in practice. The problem is that they are expensive. For example, it was found in by Braun et al. that genetic algorithms (a specific kind of

evolutionary algorithm) usually obtained better schedules than alternative methods but took up to 300 times as long to do so [13]. Another drawback of these methods is that they tend to require a lot of testing in order tweak the parameters that they use [88].

2.2 Heuristics

Heterogeneous schedulers of this type are typically divided into three broad and occasionally overlapping categories: clustering, duplication-based and listing [23, 88]. Some of these were originally proposed for homogeneous architectures and extended to heterogeneous ones, while others were explicitly created for heterogeneous systems. Regarded by Topcuoglu, Hariri and Wu as generally the most practical and best-performing category [88], we focus largely on listing heuristics in this report as they have also proven to be the most popular in practice. In Section 2.3 we describe several examples (some of which incorporate features of the other classes) in detail, all of which are then evaluated through numerical experiments in Chapter 4.

2.2.1 Clustering

These heuristics work by first clustering all tasks in the DAG according to some criterion and then scheduling each of the clusters to a single processing resource. Topcuoglu, Hariri and Wu comment that heuristics of this form tend to be impractical because the clustering is generally done by initially assuming infinitely many processors and then (if need be) merging them to match the number that are actually available, which tends to be prohibitively expensive [88]. We do not consider any examples here but the motivating idea of minimizing communication costs by assigning subsets of the tasks to a single resource influences some of the heuristics that we do discuss.

2.2.2 Duplication-based

The motivation behind schedulers in this class is to reduce communication costs by ensuring that communicating tasks are scheduled on the same resource, even if this requires them to be duplicated—i.e., the same task may be redundantly scheduled on more than one processing unit [38]. Heuristics of this type can often be superior to alternatives in terms of the quality of the schedules they obtain [33]. However there are downsides. They generally have very high time-complexity bounds [88]. Also, controlling the amount of duplication can be tricky: unnecessary duplication can lead to the system becoming clogged, with duplicated tasks obstructing the optimal scheduling of others.

2.2.3 Listing

Heuristics of this type work by first assigning a priority to all of the tasks in the DAG according to some criterion (the *task prioritizing* phase) and then allocating the tasks to the processors according to some other rule (the *processor selection* phase).

Apart from their empirical performance and popularity, another reason for our interest in listing schedulers above others is that many of them are *criticality-aware*, by which we mean they in some way attempt to take advantage of the critical path (for example, by prioritizing tasks along it above all others) in order to achieve good schedules. This idea of planning along the critical path of a DAG is closely linked to dynamic programming and its extension to problems for which our knowledge about the system is incomplete, reinforcement learning (see Chapter 5).

An additional step found in several listing (or otherwise) heuristics is to first divide the DAG into a hierarchy of *levels* such that all entry tasks are in the first level and, for all other tasks, its parents can only be found in higher levels and its children in lower ones. There are many different ways this can be done and two examples are described in Sections 2.3.6 and 2.3.5. The advantage of this approach is that the tasks in each level are independent of one another (i.e., there are no precedence constraints between them) and so they can be handled in parallel. An example is the classic Levelized Min-Time algorithm [39].

2.3 Examples of listing heuristics

In this section we describe several static listing schedulers in detail. This collection is somewhat eclectic, with the only criterion for inclusion being that they have been reported in other publications, such as [73], to be at least competitive with the best current approach (or indeed that they *are* the best). In Chapter 4 we perform our own simulated testing of all of these algorithms to evaluate their suitability for the accelerated environments that concern us in this research. In many cases, we also consider possible modifications in an attempt to improve their performance in such cases. First, however, we introduce some concepts and notation used throughout the remainder of this chapter.

2.3.1 Notation

Suppose we have an application task graph G comprising v tasks and e edges, and a heterogeneous target platform with q processing resources (which, as per Section 1.4, are either CPU cores or entire GPUs). Let w_{ij} be the estimated time it takes to execute task t_i on processing unit p_j; note that if we want to optimize anything other than execution time we can define w_{ij} accordingly, such

as estimated power consumption if we want to minimize energy usage. Then we define the *average execution cost* of task t_i by

$$\overline{w_i} \coloneqq \sum_{j=1}^{q} w_{ij}/q. \tag{2.1}$$

Let d_{ij} be the amount of data that needs to be transferred from task t_i to task t_j before it can be executed (which will of course be zero if no such dependency exists). Let b_{mn} be the data transfer rate between processing units p_m and p_n (i.e., the *bandwidth*), and $\overline{B}$ be the average of all such transfer rates, i.e.,

$$\overline{B} = \sum_{m,n} \frac{b_{mn}}{q(q-1)}.$$

Similarly, let L_m be the communication startup cost of p_m and $\overline{L}$ be the average of all processor communication startup costs.

The total communication cost from task t_i scheduled on p_m to task t_j scheduled on p_n, is then given by

$$c_{ij} \coloneqq L_m + \frac{d_{ij}}{b_{mn}} \tag{2.2}$$

and the *average communication cost* of the edge from task t_i to task t_j is defined as

$$\overline{c_{ij}} \coloneqq \overline{L} + \frac{d_{ij}}{\overline{B}}. \tag{2.3}$$

In this report we do not explicitly consider communication startup costs and thus assume $L_m = 0$ for all m. However, they can be viewed as being implicitly incorporated by the methods through which we typically generate data transfer amounts for the DAGs that we consider (see Section 3.2.3).

For all tasks t, we define $P(t)$ to be the set of all its *parent* tasks—its immediate predecessors in the DAG—and $C(t)$ to be the set of all its *child* tasks—its immediate successors.

Let $EST(t_i, p_m)$ be the *earliest start time* at which processor p_m can execute task t_i. If R_{m_i} is the earliest time at which p_m is actually free to execute task t_i and $AFT(t_k)$ is the time when execution of task t_k is actually completed (which is assumed to be known once it has been scheduled), then we have

$$EST(t_i, p_m) = \max\left\{R_{m_i}, \max_{t_k \in P(t_i)} (AFT(t_k) + c_{ki})\right\}.$$

Note that R_{m_i} is not necessarily the latest finish time of all tasks scheduled on p_m since we may be following an *insertion-based* policy that allows tasks to be

scheduled on a processor between two tasks that have already scheduled on it, assuming that task dependencies are still respected. The *earliest finish time* $EFT(t_i, p_m)$ for task t_i on processor p_m is then given by

$$EFT(t_i, p_m) = w_{im} + EST(t_i, p_m).$$

Once all tasks have been scheduled, the makespan of graph G can then be computed as

$$\begin{aligned} M(G) &\coloneqq \max_{t \in G} AFT(t) \\ &= \max_{t_e \in G} AFT(t_e), \end{aligned}$$

where t_e is an *exit* task—i.e., one with no children.

2.3.2 HEFT

One of the most popular and well-regarded of all scheduling heuristics is *Heterogeneous Earliest Finish Time* (HEFT), both because it is conceptually simple and generally performs well in practice. Canon, Jeannot, Sakellariou and Zhang compared the makespans computed by 20 task scheduling heuristics and found that HEFT was among the best, especially for randomly-generated DAGs [17]. The same study also concluded that HEFT compares very well to alternatives with regard to *robustness* (see Section 4.8.1). For these reasons, we will often use HEFT as an exemplar for the purpose of comparison with other heuristics.

Schedulers targeting homogeneous platforms often use the *upward rank* (or *bottom-level*) of a task to determine priorities, defined as the length of the critical path from that task to the end, including the cost of the task itself [88]. Essentially, the upward rank is the expected time from when the task is scheduled to when the execution of the whole DAG is complete. HEFT extends the idea to heterogeneous platforms by using the *average* computation and communication costs across all processors. More formally, for any task t_i in the DAG, we define its upward rank $rank_u(t_i)$ recursively, starting from the exit task(s), by

$$rank_u(t_i) \coloneqq \overline{w_i} + \max_{t_j \in C(t_i)} (\overline{c_{ij}} + rank_u(t_j)). \tag{2.4}$$

Note that we can also define the *downward rank* (or *top-level*) of a task, the critical path length from its source to the task in question, excluding the cost of the task itself. There doesn't seem to be any deep theoretical justification for preferring one alternative to the other but upward ranking is suggested by the authors and comparative studies have found that it is almost always the more effective choice [108].

The HEFT task prioritization phase then concludes by listing all tasks in decreasing order of upward rank, with ties broken in some consistent manner;

the authors suggests that doing this arbitrarily is probably best in practice as it avoids the costs associated with some of the alternatives that have been proposed, such as favoring the task whose successors have the higher upward rank.

Once the tasks have been prioritized, the processor selection phase moves down the list (i.e., starting with the highest upward rank) and assigns each task to the processor that is expected to complete its execution at the earliest time (with insertion). A complete description of HEFT is given in Algorithm 2.1. If

Algorithm 2.1: HEFT.

```
1  Set the computation cost of all tasks using (2.1)
2  Set the communication cost of all edges using (2.3)
3  Compute rank_u for all tasks by traversing the DAG upward, starting
     from the end task(s), according to (2.4)
4  Sort the tasks into a priority list by non-increasing order of rank_u
5  for task in list do
6  |   for each processor p_k do
7  |   |   Compute EFT(t_i, p_k)
8  |   end
9  |   Schedule task t_i on the processor p_m that minimizes EFT(t_i, p_k)
10 end
```

we have q processors and e edges in our DAG, then HEFT has a time complexity of $O(q \cdot e)$. For dense DAGs, the number of edges is usually proportional to v^2, where v is the number of tasks, so the time complexity of HEFT in this case is $O(v^2 q)$ [88].

Many variants and extensions of HEFT have been proposed since its publication. The heuristics described in Sections 2.3.3 and 2.3.4 are explicit examples and all others in this chapter are clearly at least inspired by the general approach. A dynamic version of HEFT was used in numerical experiments with their own scheduling heuristic by Chronaki et al. [23]. StarPU also implements a dynamic HEFT-like algorithm called the *deque model* (DM) scheduler which schedules all tasks as soon as they become ready on the processor with the earliest estimated finish time. Several other minor tweaks of this scheduler are also implemented in StarPU, such as *deque model data aware* (DMDA), which also takes data transfer times into account [78].

Despite its popularity, there are situations for which HEFT may encounter problems. Many of these are a result of the fact that it is intrinsically greedy, always scheduling tasks on the processor predicted to complete its execution first (i.e., solving the one-step problem optimally). This means, in particular, that it cannot avoid possible future penalties. We will see in Chapter 4 that there are many examples of realistic DAGs and (accelerated) environments for which HEFT obtains a worse makespan than simply scheduling all tasks on the single fastest

processor because it allocates tasks without considering the later communication costs this will imply. Alternative task assignment methods that look beyond the immediate problem have been proposed to alleviate this issue; see Sections 2.3.3, 4.2.3 and 4.5.

A less obvious issue with HEFT is how it extends the idea of the upward rank of a task to heterogeneous environments by using mean values, which is conceptually simple and cheap to implement but which we have no deep theoretical reason to believe is actually optimal. Alternative weightings have been considered by multiple earlier publications, with the most comprehensive study probably being Zhao and Sakellariou [108]. They compare five alternatives with the default mean value approach:

- *Median.* Rather than using mean computation and communication values, use the median instead.
- *Worst.* Use the worst-case computation costs (i.e., take all task weights to be the maximum of the CPU and GPU times in our case) and determine the communication costs based on this (so e.g., if the worst-case weighting suggests a task and its child should use CPU and GPU execution costs, respectively, then take the communication cost to be the cost of communicating between a CPU and GPU).
- *Best.* Same as previous but instead of the worst-case values, use the best instead.
- *Simple worst.* Use the worst-case computation and communication costs (so rather than determining the latter based on the former, always use the worst possible value.)
- *Simple best.* Same as previous but with best-case values instead of worst.

In addition, all six (including the mean value default) of these are considered with either upward or downward ranking, giving a total of twelve possibilities which were evaluated through extensive numerical experiments. The DAGs used were either randomly-generated (with between 25 and 100 tasks) or representative of a Laplace equation solver (with between 25 and 400 tasks). Costs were generated either via an entirely random method (selection from a uniform distribution with a predefined range) or a randomized procedure that weights the task execution costs on a given processor in proportion to the relative power of the processor. (The latter is more similar to the approach that we have taken in our own experiments, see Section 3.2.3.)

Zhao and Sakellariou ultimately concluded that the standard mean value weighting was not superior to the others and that in fact some of the alternatives were more promising—although none of them uniformly outperformed all others and all were capable of performing badly. More decisively, their experiments also

suggested that the upward ranking approach always outperformed the downward ranking alternatives for the random DAGs, although not the Laplace DAGs.

Introduced in the same paper as HEFT is *Critical-Path-On-a-Processor* (CPOP). It has not proven to be as popular and has generally been found to perform more poorly [17]. The key idea is to minimize communication costs by ensuring that all tasks on the critical path are scheduled on the same processor; other tasks are allocated to the fastest processor as in HEFT. To determine which tasks are on the critical path, task priorities are instead computed as the sum of upward and downward ranks, the idea being that tasks with the highest such sum at each "level" of the DAG will be critical path tasks [88]. Although we do not consider CPOP in any detail in this report since we believe earlier publications have established that it is not in general a promising approach, we may refer to it occasionally.

2.3.3 HEFT with Lookahead

Bittencourt, Sakellariou and Madeira proposed an extension of HEFT that incorporates *lookahead* in an attempt to avoid the difficulties it can have because of its intrinsic greediness [10]. The basic idea is that rather than scheduling a task on the resource predicted to minimize its EFT, we schedule it on the resource that minimizes the EST of its *children*. In particular, the authors consider four variants, assuming that the task prioritizing phase has already been done in the usual HEFT way:

- For each task in the priority list, choose the processing unit which minimizes the maximum EFT of all its children.
- For each task t in the priority list, choose the processor which minimizes a weighted average of the EFTs of its children, calculated for each processor p through

$$W(t,p) := \frac{\sum_{c \in C(t)} (rank_u(c) \cdot EFT(c))}{\sum_{c \in C(t)} rank_u(c)},$$

  where $rank(c)$ is the upward rank of the child task.
- If the next two tasks t_1 and t_2 in the priority list are independent, then we can swap the order in which they are scheduled if the maximum EFT of the children of t_2 is less than that of t_1.
- Same as above but a swap is made if it minimizes the weighted average of the child tasks.

As a wise man once said, it is difficult to make predictions, especially about the future, so the obvious issue with a lookahead heuristic of this type is how

exactly the EFT of the child tasks is estimated. To simplify matters, the algorithm assumes that all child tasks can be scheduled immediately after the current task has been scheduled—disregarding the possibility that the children may have other parent tasks that have not yet been scheduled—and then computes their EFT assuming that we follow the standard HEFT insertion-based fastest processor selection. The authors acknowledge this means that the estimates will be optimistic but it is admittedly difficult to think of a better alternative that is not significantly more expensive. No matter which of the four variants is used, the time complexity of the whole algorithm relative to standard HEFT is an additional factor of $q \cdot \bar{c}$, where q is the number of processing resources and $\bar{c}$ is the average number of children per task, which can be significant.

The lookahead step can of course be extended beyond just a task's children. However, based on preliminary results, the authors suggest that there is generally little advantage gained by looking further ahead. Certainly this is something we may investigate in the future but it is easy to believe, given that the one-step estimates themselves are unlikely to be accurate, that building upon them may well introduce too much uncertainty for the result to be useful.

After numerical experimentation with both real and random DAGs, the authors conclude that their new algorithm typically outperforms standard HEFT, particularly when there are lots of processing resources and the total communication costs are high relative to the computation costs. In addition, their results suggested that there was rarely any benefit to the priority list change variant compared with the lookahead alone version.

For ease of reference, we may refer to this heuristic as *HEFT-L* in this report.

2.3.4 HEFT No-Cross

In many ways the most relevant of all the heuristics introduced in this chapter is HEFT No-Cross, proposed by Shetti, Fahmy and Bretschneider, because it is an extension of HEFT specifically targeting CPU-GPU environments [73]. The algorithm has the same basic structure as HEFT but makes modifications to both the task prioritization and processor selection phases. For the former, they consider three alternative task weightings that are used to compute the upward rank:

1. the speedup, the ratio of the time on the slowest processor p_s (usually the CPU) to the time on the fastest processor p_f (usually the GPU).

2. the actual absolute value of the time difference between slowest and fastest processor.

3. the ratio of the previous two weightings, i.e.,

$$W_t = \frac{abs(w(t, p_s) - w(t, p_f))}{w(t, p_s)/w(t, p_f)}.$$

For ease of reference, we call these three weightings *speedup*, *diff* and *NC* respectively. The motivation behind all three is that since the CPU and GPU task execution times may be vastly different simply using the mean of the two may not be adequate. Note that the upward rank is computed using these task weightings alone—i.e., without considering communication costs; the idea is that the modifications to the next phase of the algorithm will implicitly compensate for this.

For the task assignment stage, the authors propose "no-crossover scheduling". To decide if a task t_i should be scheduled on the processor p_j that minimizes its EFT, we first consider the processor p_f which minimizes the execution time of t_i. If $p_j = p_f$ then we schedule t_i on p_j. If not, we compute

$$W_{\text{abstract}} = \frac{abs(EFT(t_i, p_j) - EFT(t_i, p_f))}{EFT(t_i, p_j)/EFT(t_i, p_f)}$$

and if

$$W_{t_i}/W_{\text{abstract}} \leq X,$$

where X is some constant called the cross threshold, then we schedule t_i on p_j; otherwise, we schedule t_i on p_f. The immediate issue is how we find a good cross threshold; the authors suggest $X = 0.3$ is a good value based on their numerical experiments but finding the optimal choice in general is an open question.

The authors compare their new algorithm with HEFT through simulations with 2000 randomly-generated DAGs and conclude that it usually achieves a smaller makespan, with an average reduction of roughly ten percent.

2.3.5 PETS

As already noted, many simple alternative task ranking steps have been considered for the HEFT algorithm. There are also heuristics that simply use a more complex ranking in the same framework, such as *Scheduling on Multi-Clusters* (SMC), which incorporates communication costs when computing approximate task weights but otherwise proceeds as in HEFT [58]. Another similar heuristic is *Performance Effective Task Scheduling* (PETS), which has the same processor selection phase but uses a more complex task ranking and an additional leveling phase [35].

The PETS algorithm begins by computing these levels in a straightforward way: for all tasks t, its level is defined to be the number of tasks in the longest possible chain from an entry task to t (inclusive). Once this has been done, task ranks are computed level-by-level as $rank(t) := round(ACC + DTC + RPT)$, where

- ACC (*Average Computation Cost*) is the mean execution cost across all processors,

- DTC (*Data Transfer Cost*) is the sum of communication costs with all the child tasks in the next level,
- RPT (*Rank of Predecessor Task*) is the highest rank of all its parent tasks in the previous level.

The processor selection phase then begins by dealing with the levels in ascending order. Tasks in each level are considered in descending order of rank, with ties broken by ACC (where smaller values have higher priority), and scheduled on the processor which minimizes their EFT, as in HEFT. The complexity of the full algorithm is $O(e \cdot (q + \log v))$, where e is the number of edges, q is the number of processing units and v is the number of tasks, which is broadly competitive with HEFT.

Computing the ACC and RPT is straightforward but note that there is some ambiguity in calculating the DTC on the accelerated architectures we are concerned with since the communication cost between any two tasks depends on where they are both scheduled; for example, if they are both on CPU then the cost is zero, but it will not be if one is on a CPU and the other a GPU. There are two different approaches then for computing the DTC: predicting where a task and all its children will be scheduled, or using some approximate communication cost as in HEFT. The latter is simpler since the former is both computationally expensive and probably unreliable, so this is what we have done in our own implementation of PETS (see Section 4.3).

After comparing the performance of both HEFT and PETS for a collection of both real and randomly-generated application DAGs, the authors conclude that PETS is consistently superior by most metrics that they consider, including makespan of the resulting schedule.

2.3.6 HBMCT

Another example of a listing scheduler which uses a level-based approach is HBMCT, from Sakellariou and Zhao [70]. The underlying motivation is to break the DAG up into *groups* of independent tasks and schedule these in a nearly optimal way; in some sense then it is similar to HEFT but rather than solving the one-step problem of optimally scheduling individual tasks, it attempts to do the same at the group level. The structure of the algorithm is as follows.

1. Rank all tasks. The authors use the same mean value upward ranking as in HEFT but in theory anything else can be used.
2. Divide all tasks into prioritized groups such that all tasks in each group have higher rank than all tasks in groups with lower priority. Note that these groups therefore differ from, e.g., levels in PETS, because they take task ranks into consideration, not just DAG topology. The advantage of

this is that otherwise it is possible a task with low priority (as defined by the ranking) can be scheduled too early and obstruct the scheduling of later higher priority tasks.

3. Schedule all the tasks in each group (starting with the highest priority one and so on) using any heuristic for scheduling independent tasks on heterogeneous platforms. The authors consider several existing examples from [14] and also propose a novel heuristic called *Balanced Minimum Completion Time* (BMCT) which works by initially scheduling all tasks in the group to their fastest processor and then iteratively redistributing those tasks among the processors in order to minimize the current makespan. The idea being that the final scheduling of each group will then be close to optimal, given that the tasks in groups with higher priorities have already been scheduled, and that although this would be far too expensive for the DAG as a whole, the groups will hopefully be small enough that the cost will not be prohibitive.

Numerical experiments from the original paper and elsewhere suggest that in general HBMCT is at least competitive with HEFT and other similar heuristics [17].

The immediate issue with HBMCT is the cost of the task rearrangements in the processor selection phase; the authors note that although it outperformed all of the alternative independent task scheduling heuristics they considered, it was also the most expensive. So the question then is whether it is possible to determine a priori, for a given DAG, that this extra cost is justified by significant reductions in schedule makespan, since there is generally no guarantee that solving the group-level subproblems optimally will actually lead to a smaller makespan overall anyway. The key may be to select a suitable initial task ranking that increases the likelihood that this is the case; we investigate some possible choices in Section 4.7.

2.3.7 PEFT

To incorporate a degree of lookahead into the HEFT framework while also maintaining the same time complexity—i.e., quadratic in the number of tasks—Arabnejad and Barbosa propose the *Predict Earliest Finish Time* (PEFT) heuristic [3]. The main innovation is that rather than just considering the EFT of a task during scheduling as in HEFT, or estimating the finish time of its children as in HEFT-L, we instead attempt to minimize an optimistic estimate of the time it will take to execute the entire DAG once the task has been scheduled, assuming that there are infinitely many processors available. More precisely, for all tasks t and processors

p the algorithm begins by computing the *optimistic cost* recursively through

$$OC(t,p) = \begin{cases} 0, & \text{if } t \text{ is an exit task,} \\ \max_{s \in C(t)} \left\{ \min_{p'} \{ OC(s,p') + w_{tp} + \bar{c}_{ts} \} \right\}, & \text{otherwise,} \end{cases}$$

where the average communication cost between tasks t and s is taken to be zero if $p' = p$. Note that the recursive definition means that only the immediate children of a task need to be considered.

Once all optimistic costs have been computed, tasks are prioritized using these values. The authors suggest that for all tasks, the mean of the optimistic costs across all processors should be used, with higher values being ranked first. Unfortunately this does not always ensure that task priorities respect precedence constraints (see example in Section 4.6). At any rate, there are other ranking methods based on the optimistic cost that we can use. The most natural choice would be to define the rank of a task to be the minimum optimistic cost over all processors, as this will ensure that precedence constraints are maintained. The processor selection phase of PEFT then proceeds by scheduling the tasks in order of their priority to the processor which minimizes the sum of their EFT and OC, i.e., for all tasks t and processors p we compute

$$O_{EFT}(t,p) \coloneqq EFT(t,p) + OC(t,p)$$

and schedule tasks on the processor which minimizes this value.

Simulations with random and real application DAGs detailed by the original authors suggest that PEFT is generally superior to HEFT in terms of schedule makespan, with the curious exception of Fast Fourier Transform graphs; this discrepancy was attributed to the fact that for such graphs, all tasks lie on the critical path.

2.4 The effect of stochasticity

Even when static, deterministic scheduling heuristics are used in practice, the expectation is almost never that the values they use will actually be totally accurate. Instead, the assumption is that these will be close enough that the computed schedules will be still be useful for the real application DAGs with the "true" weights, while also being cheaper and easier to deal with than modeling the problem stochastically. So what we want is for a task scheduling heuristic to be *robust*, with small changes in DAG weights leading only to correspondingly small changes to the quality of their schedules. Previous publications have investigated the robustness of static scheduling heuristics [17], and we do the same for all the examples from this chapter in Section 4.8.1.

Many static listing heuristics have been modified for the stochastic DAG scheduling problem. For example, Canon and Jeannot propose Rob-HEFT, an

extension of the HEFT heuristic that attempts to find the ideal tradeoff while attempting to minimize both the schedule makespan and its standard deviation. A more straightforward extension is *stochastic heterogeneous earliest finish time* (SHEFT) from Tang et al., which takes the expected values of the costs to be the DAG weights and then proceeds as in the deterministic algorithm [82]. Several authors of the latter, with others, later proposed the more sophisticated *stochastic dynamic level scheduling* (SDLS) heuristic, based on the *stochastic bottom level*, which they define for all tasks to be the maximum length of a longest path to an exit task [51].

Given the difficulty of estimating the distribution of schedule makespans for the stochastic scheduling problem analytically, a popular alternative is to use Monte Carlo simulation to generate realizations of the makespan and approximate its distribution from these, an idea that dates back to at least the early 1960s [93]. A more recent example is the *Monte Carlo-based Scheduling* (MCS) heuristic from Zheng and Sakellariou, which works by building on schedules computed by any standard deterministic heuristic (such as the ones we have described in this chapter) and using Monte Carlo sampling from the task computation time distributions to determine a more robust schedule [110]. Simulations performed by the authors suggested that their Monte Carlo approach always outperformed heuristics that find a schedule directly from the DAG itself. With regards to this report, the important takeaway from this is that static schedules can still be useful even for the stochastic problem. In particular, since the makespan of the stochastic schedule depends to an extent on the makespan of the initial static schedule, determining good static scheduling heuristics can be important even in that case.

Chapter 3

Testing environment

To evaluate how the existing task scheduling heuristics from Section 2.3 perform for the accelerated computing environments we are interested in, we created a simple software framework that allows us to simulate the scheduling and execution of DAG applications on user-defined accelerated computing platforms. The simulator is written in `Python` and the source code in its entirety can be found at the following Github repository:

`https://github.com/mcsweeney90/python-simulator`.

More established and sophisticated software already exists for this purpose. In particular, we considered the use of SimGrid [19] because of its comparability with StarPU and the continuing collaboration between the two development teams [76, 77]. (The idea being that users can create efficient application code with StarPU and simulate its execution on arbitrary target platforms with SimGrid.)

However, we ultimately decided to implement our own simulation environment in the hope that this will give us more flexibility to experiment and, on a personal level, allow us to develop a more "bottom-up" understanding of the problem. (Working in `Python` also allows us to more easily take advantage of the many excellent machine learning packages which have been developed in that language, such as `Keras` [22], which will be very useful when we begin to investigate the possibility of applying reinforcement learning to the task scheduling problem—see Chapter 5 onward.) Once we have successfully prototyped promising approaches using our own simplified simulator, our hope is to then migrate to more realistic simulation software—and ultimately real HPC systems—in future work.

3.1 Simulated environments

The target architectures for this research are comprised of multicore CPUs and some accelerator(s), most likely GPU(s). So throughout the remainder of this report, we often consider individual nodes of two state-of-the-art HPC machines:

1. Titan: 18,688 nodes, each with a 16-core AMD Opteron CPU and a NVIDIA

Kepler K20X GPU, linked via a Cray Gemini interconnect [46].

2. Summit: 4,608 nodes, each with two 22-core IBM Power9 CPUs and six NVIDIA V100 GPUs, linked via dual-rail Mellanox EDR Infiniband interconnect [45].

We focus on individual nodes because these capture in microcosm the fundamental issues raised by scheduling on heterogeneous platforms; at a node level, both of these machines are homogeneous. We chose these two examples in particular in order to consider scheduling in both single and multiple GPU environments.

While we aim to represent these environments as accurately as possible in our software, our simulated representation is of course a vast simplification. In future work we intend to evaluate the quality of schedules obtained through simulation on their actual target platforms.

3.2 DAGs

In order to accurately evaluate any heterogeneous scheduling algorithm through simulation, we need a body of suitable DAGs to test them on. All results presented later in this report are based on DAG topologies generated through the methods described in the following two sections. For a given topology, we then generate realistic task computation and communication costs through the procedure described in Section 3.2.3.

3.2.1 NLA DAGs

In task-based numerical linear algebra applications, tasks are usually BLAS calls on tiles of a given matrix and DAGs representative of such applications are fairly easy to construct from the underlying algorithms themselves. For example, Algorithm 3.1 gives a rough description of a Cholesky factorization of a 2×2 tiled matrix A; the extension to finer matrix tilings is done in the natural way. Note that we make use of only four BLAS routines in total (and thus have tasks of only four different types): `GEMM` (matrix multiplication), `POTRF` (Cholesky factorization), `SYRK` (symmetric rank-k update) and `TRSM` (triangular solve).

The function `cholesky` in our simulator source code allows us to construct Cholesky-like DAGs of arbitrary size. We assume that the matrix is divided into square tiles of equal size and so as input the function takes the number of tiles along either axis and the tile size. Figure 3.1 is the DAG formed using this method for a 3×3 block matrix A. Of course, like any algorithm, many different implementations of Cholesky factorization are possible. This is a fairly typical example of how it is usually done in practice today but it should always be borne in mind that when we present results for Cholesky factorization DAGs, for example, we really mean DAGs which represent this particular implementation. (One

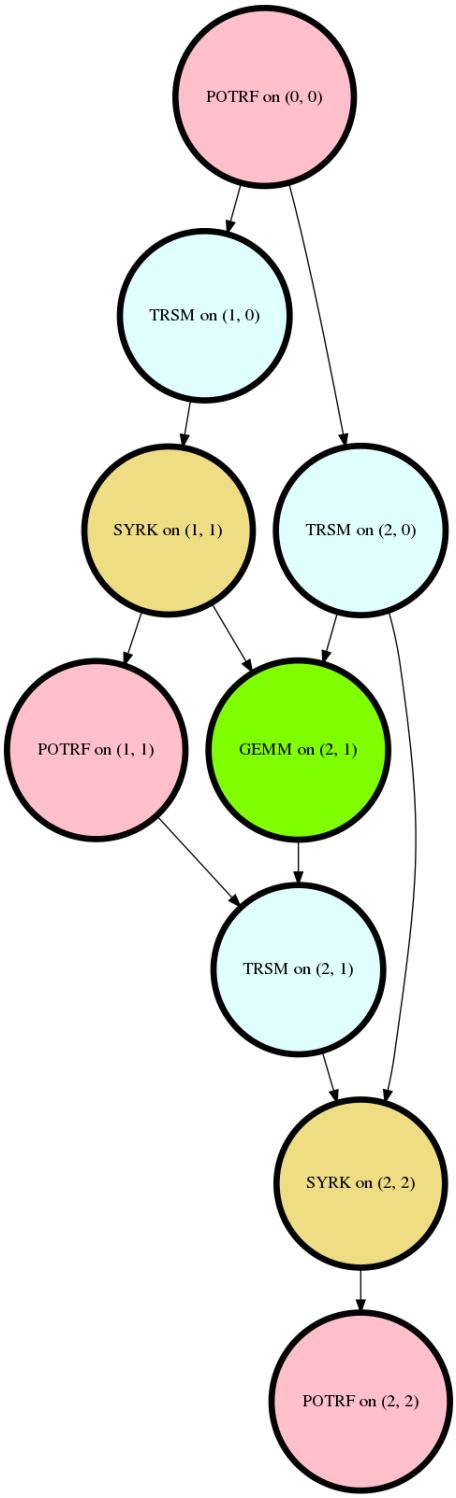

Figure 3.1: Cholesky factorization of a 3×3 tiled matrix A. Labels specify the routine and tile it is called on.

Algorithm 3.1: A practical algorithm for computing the block Cholesky factorization of a matrix using BLAS routines.

Input : Symmetric positive definite matrix $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{12} & A_{22} \end{bmatrix}$.

Output: Lower triangular matrix L such that $A = LL^T$. For efficiency, the matrix A is overwritten such that $A = \begin{bmatrix} L_{11} & \\ L_{12} & L_{22} \end{bmatrix}$.

while *A is not fully factorized* **do**
  Update $A_{11} \coloneqq L_{11} = \texttt{POTRF}(A_{11})$
  Compute $A_{12} \coloneqq L_{12} = A_{12}L_{11}^{-T}$ using `TRSM`
  Compute $A_{22} \coloneqq L_{22} = A_{22} - L_{21}L_{21}^T$ using `SYRK` and `GEMM`
  $A \coloneqq A_{22}$
end

of the reasons why we believe that reinforcement learning is a promising avenue for task scheduling is because of the potential for transfer of learning—which has long been a central aim in that field—between, for example, similar DAGs which represent different implementations of the same underlying algorithm. See Section 7.5.)

3.2.2 The Standard Task Graph (STG) set

Tobita and Kasahara [84] propose a set of task graphs for benchmarking scheduling algorithms and heuristics, called the Standard Task Graph set, which are freely available online [41]. This is a collection of 2700 task graphs, 180 each of a given size (i.e., the total number of tasks) from 50 to 5000, with additional artificial zero-cost entry and exit tasks (so actually between 52 and 5002 tasks in total)[1]. The DAGs were generated via several different standard methods with the intention of capturing as wide a variety of features (degree of parallelism, density, etc) as possible. Also included as part of the STG are three DAGs from real applications, one from a robot control problem, one from a sparse matrix solver, and the other from a standard `FORTRAN` benchmark.

Although the DAGs in the STG include randomly-generated computation costs, we only use the DAG topology itself in our experiments as we prefer to generate our own in an attempt to ensure that their values on the two processor types are realistic relative to one another; our approach is described in detail in the following section.

[1]Note that we typically assign non-zero weights to the entry and exit tasks in our experiments.

3.2.3 Generating realistic weights

Typically when a user wishes to execute an application, computation and communication times will be estimated in one of two ways:

1. Theoretical arguments. For example, in numerical linear algebra, since BLAS flop counts and peak processor speeds (in flops) are usually known we can compute lower bounds for BLAS timings on a given processor by simply dividing the former by the latter. This lower bound can then be used as a (very optimistic) estimate of the computation time.

2. Previous execution data. For example, if a task has been executed on a processor one hundred times before, we could simply use the mean of those timings as an estimate. Typically runtime systems such as StarPU which predict task execution times based on existing data use simple statistical techniques such as linear regression.

Our simulation software permits both approaches, although at present our method for the latter is very unsophisticated, computing the mean and standard deviation of the input timing data for a given task and estimating its cost as a random variable from a normal distribution with these parameters. We intend to incorporate more sophisticated methods into the software in future. Since we lack relevant execution traces for the architectures we are interested in, we will typically use an extension of the first approach for predicting DAG weights for the numerical results presented in this report.

As mentioned, flop counts for BLAS implementations are generally well-known and so defaults for many routines are included as part of the simulation software (e.g., $2t^3$ for `GEMM`, where t is the tile size). Likewise, theoretical peak performance estimates in flops per second[2] for all the processor types we consider in this report are also included (calculated through clock frequency $\times$ flops per cycle). So for all the NLA DAGs we consider, when tasks are BLAS calls on tiles of known size, we can compute a lower bound on the task computation costs by dividing the flop count by the peak processor performance. This lower bound is extremely unlikely to be achieved in practice so, in an attempt to make these costs slightly more realistic, the simulator allows users to "add some noise"—specify upper and lower bounds (ℓ, u) such that the execution time of a task is actually set as $r \cdot \hat{w}(t)$, where $\hat{w}(t)$ is the theoretical time computed using flop counts and peak processing speeds and r is chosen uniformly at random from the interval (ℓ, u). Unless otherwise stated, for all results presented in this report we take $\ell = 1$ and $u = 10$ on both CPU and GPU. The idea being that the execution time will then be within an order of magnitude of the lower bound and, more importantly, the

[2]We actually use flops per microsecond to make the values more human-friendly but this is applied consistently throughout so makes no difference.

ratio of the CPU and GPU times will be roughly what would be predicted by their speeds (and also not identical for all tasks).

We follow largely the same procedure for the the randomly-generated DAGs from the STG but as the flop counts must be generated, we assign to each task a flop count chosen uniformly at random from an integer range calculated to restrict the GPU time lower bounds to the interval $(1, 100)$, then compute the lower bounds on the CPU and GPU times using this flop count. The actual execution times are then calculated by multiplying the lower bound by a factor chosen uniformly at random in the same way that we do for the NLA DAGs.

The immediate objection to our method for generating task weights is that we have implicitly assumed that task times are determined entirely by processor speed. This is of course reductive and potentially even harmful since the most interesting situations are arguably those for which timings on the different processor types do not follow such a simple pattern. The most common alternative approach is to instead assume that the ratio of CPU and GPU timings is a random variable from some distribution. For example, Canon, Marchal, Simon and Vivien (who also use the STG) generate costs by randomly choosing GPU times and then estimating the corresponding CPU times through multiplication with a random variable from a gamma distribution with expected value 15 and standard deviation 15, values suggested by numerical experiments with linear algebra kernels using the Chameleon software package [18]. However, our concern is that this method also in some sense introduces bias since it is possible that timings for different applications may not follow the same distribution. In future work we intend to consider how using real execution data in this manner compares to our chosen approach. (Note also that the method of Canon, Marchal, Simon and Vivien is implemented as an alternative way to generate task execution times in our simulation software.)

For NLA application DAGs, we assume that the data to be transferred between a parent task and each of its children is simply the size (in bytes) of the tile on which the parent acts. (The simulator software has an optional precision argument used to specify the size of each entry of the tile, with `double`—8 bytes—being the default.) The communication cost between the tasks is then calculated by dividing the data transfer by the interconnect bandwidth. Peak theoretical bandwidths for several common interconnects are included as defaults in the simulator software and, although actual effective bandwidth rarely matches this in practice, these are the values used for all numerical results presented in this report, unless explicitly stated otherwise.

To generate data transfers (and therefore communication costs) for DAGs from the STG we first assume that all tasks must transfer the same amount of data to all of their children. This is reasonable for NLA applications although not necessarily more general ones. However, we do not believe this restriction will significantly affect the veracity of our results given our procedure for randomly generating these values, which is as follows. For each task in the DAG we asso-

ciate a randomly chosen data amount (in bytes) to be transferred to its children, according to two possible *regimes*: one, that all communication costs are of the same order as the mean GPU time ("low" data), and two, that all communication costs are of roughly the same magnitude as the mean CPU time ("high" data). The low data regime is more realistic for NLA applications given that tile sizes are usually fairly small. Indeed, the high data regime is arguably less realistic even for more general applications and could be regarded as a pathological "edge case". Still, it is by no means completely unreasonable, particularly when effective bandwidth is very low. From our perspective these two data regimes are interesting because markedly divergent behavior can be observed for task scheduling heuristics in the two cases (see Chapter 4).

As stated in Section 1.4, one of the distinguishing features of this research is that we only consider data movement costs between CPU and GPU (or distinct GPUs), the assumption being that data transfer times between CPU cores are either actually zero or at least negligible relative to moving data between different processor types. So in all our experiments we only actually need to compute data transfer times in the latter case. In future work, we may consider alternative models for which this is not the case, such as when CPU and GPU are on the same physical chip (when all communication costs may be negligible).

Chapter 4

Numerical investigation of existing heuristics

In this chapter, we investigate the performance of the listing heuristics described in Section 2.3 through simulation using our custom simulator, and consider the effect of some possible optimizations. Many of these have already been studied extensively but much of that research considered arbitrary heterogeneous target platforms; our aim is to establish how they perform for CPU-GPU architectures, in particular the Titan and Summit nodes described in Section 3.1. All numerical experiments presented in this chapter were performed on a machine with an Intel m3-6Y30 quad-core processor, running at 0.90 GHz with 8GB of memory; the operating system was Ubuntu 18.04.

4.1 Performance metrics

There are many different metrics that have been used in the literature to determine how well scheduling heuristics perform but ultimately, when evaluating any algorithm, the things we really want to know are how good the solution is and how long it took to get it. For our purposes, the former is determined entirely by its makespan, which we of course seek to minimize. In some related work, schedule quality is also determined by other metrics which assume that we also want to minimize processors downtime (the idea being that any time a processor spends idle must intrinsically detract from optimal performance). This is natural in some contexts, such as when we have a continuous flow of independent tasks, but not DAG scheduling where some idle spells may be necessary even for optimal schedules because one or more tasks must wait for all their predecessors to complete before they can be executed, or because large communication costs must be avoided.

Of course, to determine how good a given schedule's makespan actually is, we need something to compare it with. The two most popular metrics therefore are

the following.

- *Speedup*, the ratio of the *minimal serial time* (MST)—the minimum execution time of the DAG on any single processor—to the makespan.
- *Schedule length ratio* (SLR), the ratio of the makespan to the critical path of the DAG (which, as previously noted, is a lower bound).

Roughly speaking, speedup tells us how much better the schedule is compared to a far simpler (and cheaper) alternative and SLR tells us how good it is relative to the best possible solution. Note that we want a *large* speedup but a *small* schedule length ratio.

In this report, we regard any heuristic producing a schedule with a makespan that does not at least match the minimal serial time as having outright *failed* for the DAG in question, since this is obviously adverse behavior for any multiprocessing environment. Note though that it can often be the case for accelerated platforms, especially for small DAGs and when the accelerator type is far more powerful than the CPU type, that the minimal serial time is actually the best possible.

With regards to the schedule length ratio, while we can see that it is clearly useful, especially for comparing two or more schedules, it should be emphasized that the critical path is only a lower bound and may be quite loose in practice, particularly for very large DAGs or, more generally, when the number of tasks vastly exceeds the available processing resources.

Speedup is generally straightforward to compute but the SLR is slightly more complex because of ambiguity surrounding how we define the critical path. For homogeneous platforms, DAG weights are effectively fixed so the critical path can be computed in any standard way (e.g., dynamic programming). But for heterogeneous platforms there are multiple possible task weights and edge weights (i.e., communication costs) may depend on where the relevant tasks are scheduled. The most common solution in the literature is to assume that all tasks can be executed without contention on the kind of processor that minimizes their execution time (i.e., set all task weights in the DAG to their smallest possible value) and then solve for the critical path. There is still some ambiguity here surrounding what edge weights/communication costs should be used. Sometimes they are simply neglected altogether, leading to a looser lower bound on the makespan. The most reasonable approach for accelerated environments is probably to set the communication cost between two tasks to be the best (i.e., smallest) possible communication cost between two processors of the kind indicated by the (already set) weight of those tasks (so if t's minimal computation time is its CPU time and s's GPU, then the communication cost would be the smallest possible time it takes to move t's data from a CPU to a GPU).

Another possible way to compute the critical path is to take processor availability into account by approaching the problem from the other direction. Instead

of essentially assuming that we have as many processors as needed to ensure that all tasks are executed on their preferred processor, we consider all possible paths through the DAG individually and find the optimal way to schedule it on the processors we actually have. However there are two major problems with this approach. First, each individual path is itself also a DAG so scheduling it optimally is easier than the original problem but still not trivial. Secondly, finding all paths through a DAG is expensive, especially when it is large and dense.

In our simulator software, the function `critical_path` is based on the first option, computed by in a classic dynamic programming manner reminiscent of the "optimistic cost" from the PEFT heuristic (see Section 2.3.7). More specifically, for all tasks t and processors p we compute the *optimistic finish time* (OFT) of t assuming that it is scheduled on p through

$$OFT(t,p) = w(t,p), \quad \text{if } t \text{ is an entry task,}$$
$$OFT(t,p) = w(t,p) + \max_{s \in P(t)} \left\{ \min_{p'} \{ w(s,p') + c_{st} \} \right\}, \quad \text{otherwise.}$$

The critical path cp is then given by

$$cp = \max_{t} \left\{ \min_{p} \{ OFT(t,p) \} \right\}, \quad \text{where } t \text{ is an exit task.}$$

All results presented in this report that refer to the critical path have been computed in this way.

Although obviously important, we do not consider the efficiency of the respective heuristics when making comparisons in this chapter, with one exception (see Section 4.5). For the most part this is because the complexity time bounds were given in Section 2.3 but also because our implementations are in no sense optimal so comparisons would be unfair.

4.2 HEFT

Previous research has established that HEFT generally performs well for a wide range of DAGs and computing environments. Our numerical experiments suggest that while this is largely true for accelerated platforms in particular, there are also a significant number of instances in which HEFT does not do so.

4.2.1 Performance

Figure 4.1 shows the speedup and SLR achieved by HEFT for DAGs representing the Cholesky factorization of $t \times t$ tile matrices on individual nodes of Titan and Summit, where $t = 5, 10, 15, \ldots,$ or 40. Task CPU and GPU times were computed as described in Section 3.2.3. We make the following comments:

- HEFT successfully returns a smaller makespan than the minimal serial time in all cases.
- Tile size makes almost no difference to either of the two metrics considered.
- The shape of the respective plots for the two different environments are almost identical, although the absolute values themselves may differ.
- The speedup on Titan is initially one because the optimal strategy is to schedule all tasks on a single GPU and HEFT successfully matches this.
- Interestingly, given that Summit nodes have six GPUs to Titan's one, the speedup of the former is generally about six times the latter and the SLR about one-sixth.
- Improvement in the speedup stagnates after the number of tasks reaches about $1,000$. It is not immediately apparent why this is so. One possibility is that due to the structure of the DAG only a certain ratio of the tasks can ever be scheduled on the CPUs without deprecation.
- For both environments, the SLR continues to grow at a roughly linear rate. It is however difficult to conclude that the performance of HEFT worsens as the size of the DAG increases since an alternative explanation is that matching the critical path becomes increasingly unrealistic as the number of tasks grows relative to the (fixed) number of processors.

To evaluate how well HEFT performs for arbitrary DAGs we make use of the STG. Table 4.1 summarizes how HEFT performs on DAGs with 502 tasks on a single node of both Titan and Summit. Computation times were again generated as described in Section 3.2.3 and we consider both "low" and "high" data variants. The most interesting takeaway is the pronounced difference in speedup for the low and high data versions; on both platforms, HEFT outright failed more often than not in the latter case.

The reason for this sharp decline in performance is that in the high data case, the minimal serial policy (i.e., the policy of scheduling all tasks to the fastest processor) is already very close to (if not identical to) the optimal policy and so superior policies are concentrated in a very narrow band around this. Because of its intrinsically greedy nature, HEFT may schedule a task without considering the necessary future communication costs that this will incur, which may result in the policy leaving this narrow band. This effect can be seen even in the low data case when the DAGs are very small, since the minimal serial policy is very close to optimal then as well. Of course, it may well be argued that such high data costs are unlikely in real applications. Still, given that HEFT was intended for use with arbitrary DAGs and platforms one would hope that it was capable of at least matching the minimal serial time in such instances. Ultimately, this is

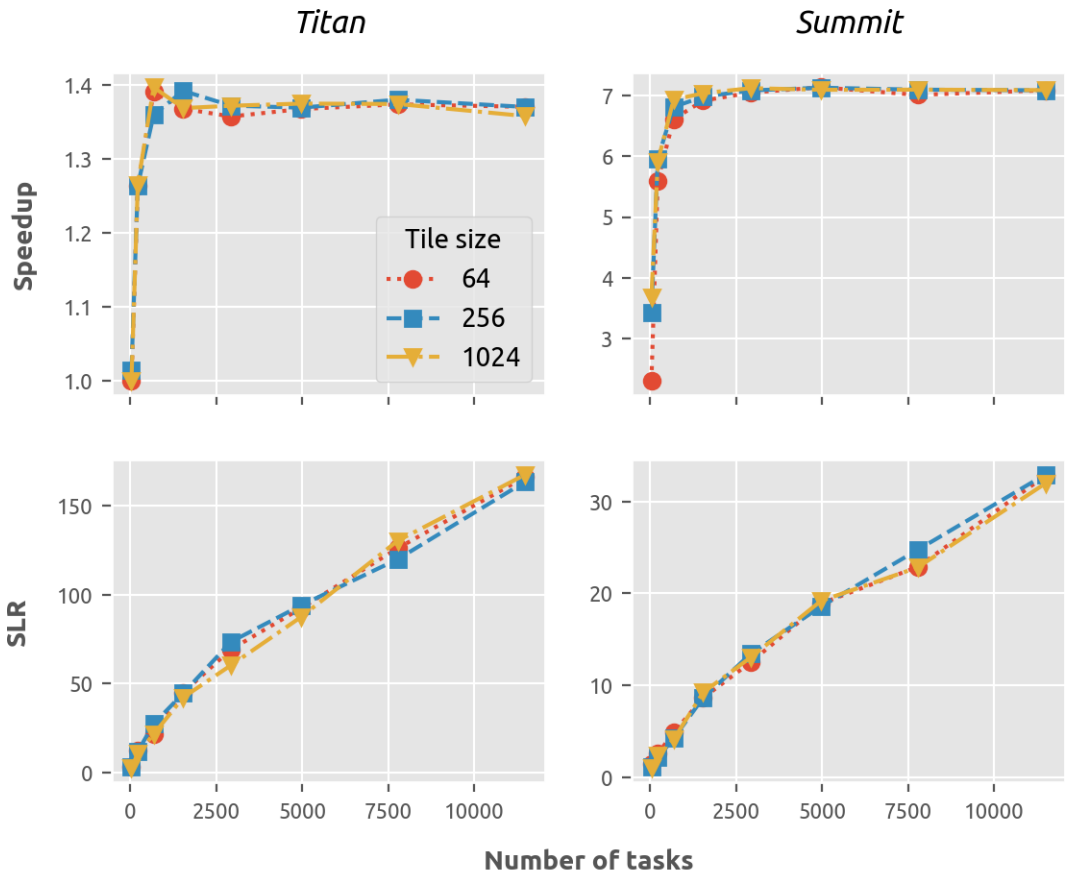

Figure 4.1: HEFT performance for Cholesky DAGs.

Table 4.1: Performance of HEFT for size 502 DAGs from the STG.

	Speedup (%)			SLR			Fails
Data	Worst	Mean	Best	Worst	Mean	Best	(/180)
			Titan				
Low	1.1	11.8	29.9	42.6	8.8	2.7	0
High	-70.1	-2.4	24.0	42.5	10.7	2.8	99
			Summit				
Low	57.3	79.1	84.9	7.6	1.9	1.1	0
High	-128.1	-25.5	42.5	34.7	12.5	2.9	133

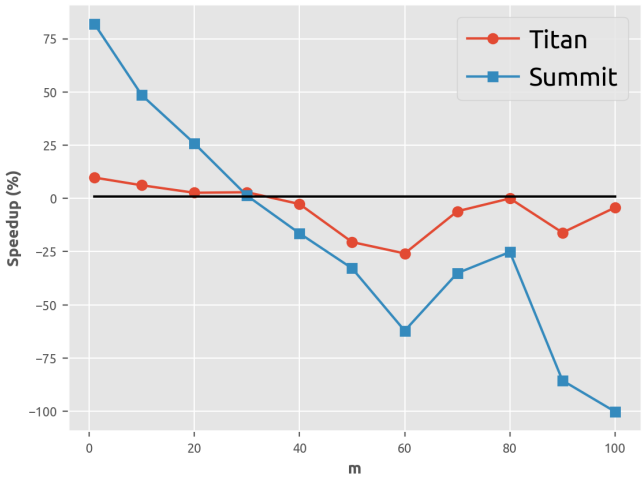

Figure 4.2: HEFT speedup for increasing m values, `rand0024` (size 502) from the STG.

a clear example of a situation in which HEFT may not be well-suited to modern accelerated computing environments.

Another interesting trend in the data is that the SLR was very similar for both data regimes on the Titan node but not the Summit node. The most straightforward explanation for this is that the multiple GPUs on Summit node lead to many more potential communication penalties that HEFT cannot avoid because of its greedy nature.

`rand0024` is a typical example of a DAG for which HEFT failed on both Titan and Summit under the high data regime. In the low data case, the HEFT makespan represented a 10% and 82% reduction, respectively, from the minimal serial time on the two platforms. However, when data amounts were high, the HEFT makespan was 5% and 21%, respectively, longer than the MST. To investigate more precisely how increasing the average communication cost affects the performance of HEFT relative to the MST, we consider a sequence of integers $m = 10, 20, \ldots, 100$ such that the average communication cost of `rand0024` is roughly m times the mean CPU time of all tasks. Figure 4.2 is the result. We see that for both platforms HEFT may fail once m reaches about 30, consistently on Summit and occasionally on Titan. Perhaps surprisingly, the speedup does not decline linearly with m and larger values sometimes achieve a superior speedup to smaller ones, so it doesn't appear to simply be the case that the algorithm is incurring the same penalties through the same actions every time.

The big conclusion from our experiments in this section is that good performance of HEFT is very much dependent on data movement costs and can often struggle when communication times are large relative to the computation costs.

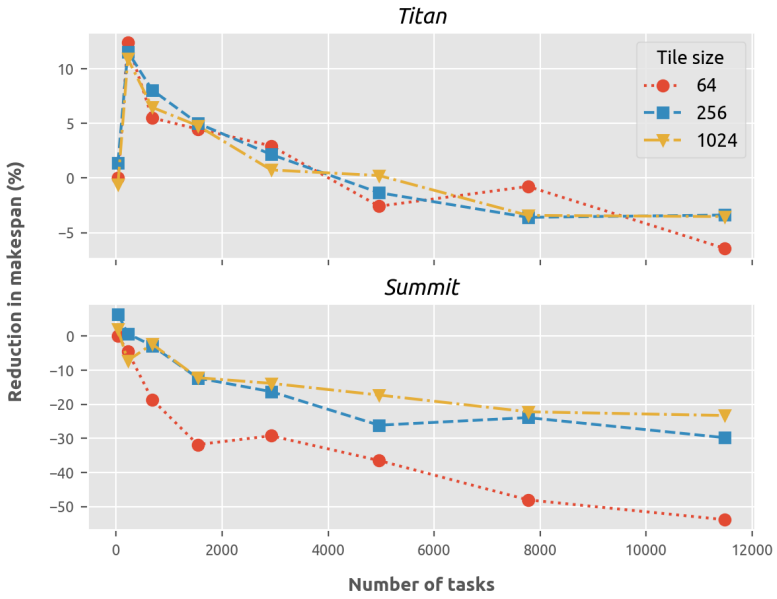

Figure 4.3: HEFT compared with RTS for Cholesky DAGs.

4.2.2 Alternative task prioritization strategies

In this section we investigate how important the task prioritization stage of the HEFT algorithm actually is and whether there are alternatives that can lead to better overall performance.

A *topological sort* of a DAG—an ordering of the tasks which respects precedence constraints—can be computed by the `Networkx` function `topological_sort`. The algorithm which this implements does not consider any objective other than ensuring that the precedence constraints are met so we can regard this as being *random* in the sense of being a random sample from the set of all possible topological orderings. Hence to gauge the importance of the task prioritization phase of HEFT, we compare the makespan obtained with the standard ranking step with that which would be obtained when using HEFT with the priority list returned by `topological_sort` instead. Although it is not truly distinct from HEFT, we call the heuristic defined by following the latter procedure *random topological sort* (RTS) from now on for ease of reference.

Figure 4.3 compares the respective makespans achieved by HEFT and RTS for a set of Cholesky DAGs on both the Titan and Summit nodes. The results are perhaps surprising, in that HEFT performs poorly, especially on the Summit node. It is not obvious why this disparity in performance for the two environments exists. Nevertheless, although the threshold at which this occurred differed for the two platforms, in both cases HEFT performance was consistently worse than RTS after the DAGs reached a certain size.

Table 4.2: % reduction in makespan of HEFT compared with RTS for DAGs from the STG. *Loses* denotes the number of DAGs (/180) for which HEFT did worse than RTS.

	Titan				**Summit**			
Data	Worst	Mean	Best	Loses	Worst	Mean	Best	Loses
502 tasks								
Low	0.1	6.5	17.1	0	-4.1	6.9	19.0	15
High	-28.2	6.4	39.1	49	-127.9	6.4	54.9	46
1002 tasks								
Low	0.6	7.5	17.7	0	-4.5	4.8	16.3	40
High	-27.2	6.8	33.6	47	-82.9	2.5	50.0	61

To consider more general applications, we repeated the comparison for DAGs from the STG with either 502 or 1002 tasks. As before, we consider both low and high data regimes. Table 4.2 displays the results. Again we see that HEFT struggled in the high data regime for both platforms compared to the alternative. More pertinently, we see that HEFT often struggled on Summit in the low data regime as well, obtaining an inferior makespan for nearly ten percent of the smaller DAGs and more than twenty percent of the larger ones, both of which are obviously undesirable. Overall, we conclude that there are fairly realistic combinations of DAGs and target platforms for which the HEFT task prioritization phase is inadequate.

Almost all of the task prioritization heuristics discussed in this report so far have the same basic structure: construct an "approximate" DAG with node/task and edge/communication weights computed in some manner and then determine task priorities by calculating their upward ranks using this approximate DAG. In standard HEFT, mean values are used to compute the approximate weights but many alternatives have been proposed. In particular, we investigated the *median*, *best*, *worst*, *simple best* and *simple worst* weightings which were previously evaluated by Zhao and Sakellariou (see Section 2.3.2) We did not consider the use of downward instead of upward ranking because of its unpromising performance in Zhao and Sakellariou's simulations, although we may revisit this decision in future work.

Figure 4.4 shows how HEFT performs with these alternatives relative to the default mean value weighting for Cholesky DAGs with tile size 64; results were similar for all other tile sizes we considered. As we may expect given that there are only two possible weights for each task and edge in the DAG, the *median* and

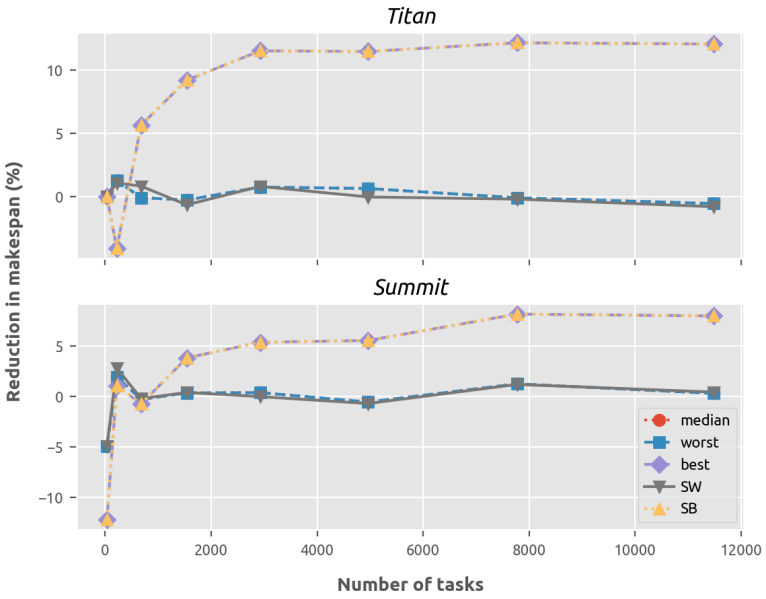

Figure 4.4: Five simple alternative weightings, Cholesky DAGs (tile size 64).

worst rankings are always identical, with *simple worst* also being very similar. Likewise, *best* and *simple best* are very similar to one another. Of these two groups, the latter was the most impressive, particularly for large DAGs, always improving on the standard mean value approach.

The same trends are not apparent when we consider DAGs with 502 tasks from the STG. In this case the metric we used for evaluation was the *average percentage degradation* (APD), the mean percentage increase in makespan of each ranking relative to the best performing ranking for each DAG. Table 4.3 shows the APD and the WPD (*worst percentage degradation*), as well as the *wins*—number of times it achieved the smallest makespan—and *fails*—the number of times it exceeded the MST—for four different task prioritization weightings. Note that we have omitted *median* from the table as it was always identical to *worst*, and *simple best* as it was always identical to *best*.

We make the following comments:

- It is difficult to draw any conclusions in general since the data are highly variable. For example, *best* (and *simple best*) were clearly the worst performing methods under the low data regime on Titan but also the best on Summit. The only conclusion we seem to be able decisively make from both the table and Figure 4.4 is that none of the weightings are uniformly superior to the others under all data cost regimes and DAG sizes (as was also the conclusion of Zhao and Sakellariou).

- Since the Cholesky DAG falls into the low data category, the respective

Table 4.3: Comparison of HEFT with four different task weightings for 180 DAGs of size 502 from the STG.

	Titan				Summit			
Weighting	APD	WPD	Wins	Fails	APD	WPD	Wins	Fails
Low data								
mean	0.4	3.6	60	0	1.6	8.1	36	0
best	2.1	7.1	9	0	0.4	3.7	103	0
worst	0.3	4.2	65	0	1.8	7.4	22	0
simple worst	0.3	4.6	59	0	1.8	6.9	19	0
High data								
mean	1.9	44.1	103	99	4.6	52.5	81	133
best	8.4	48.2	42	125	13.0	58.4	47	136
worst	8.2	52.6	45	119	12.1	57.8	40	129
simple worst	4.9	36.9	50	113	8.7	54.6	50	132

performance of the weightings seems contrary to Figure 4.4, although the superior performance of *best* (and *simple best*) for Cholesky on both nodes did not become apparent until the DAGs grew much larger. Indeed, we believe that the efficacy of scheduling heuristics for very large DAGs has been somewhat neglected in the literature, with the assumption often being that the behavior observed for smaller examples will simply scale up.

The assumption underlying the idea of using mean values as the approximate DAG weights in the standard HEFT algorithm is that, before execution, the likelihood of a given task being scheduled on all of the processors is the same, or at least that more accurate estimates are prohibitively expensive to obtain. The latter may be true in a generic heterogeneous environment with many different processor types but in the accelerated environments we are interested in it may be that we can compute further useful information relatively cheaply. This basically motivates the three alternative ranking methods *speedup*, *diff* and *NC* proposed by Shetti, Fahmy and Bretschneider [73] and described in Section 2.3.4. From their own numerical experiments, the authors suggest that all three of these lead to superior performance over the default when used in HEFT, with the third appearing to be the best.

Figure 4.5 shows how HEFT performs with these alternatives relative to the default mean value weighting for Cholesky DAGs with tile size 64; results were similar for all other tile sizes we considered. We see that indeed *NC* does appear to consistently be the best of the four, outperforming both alternatives and the

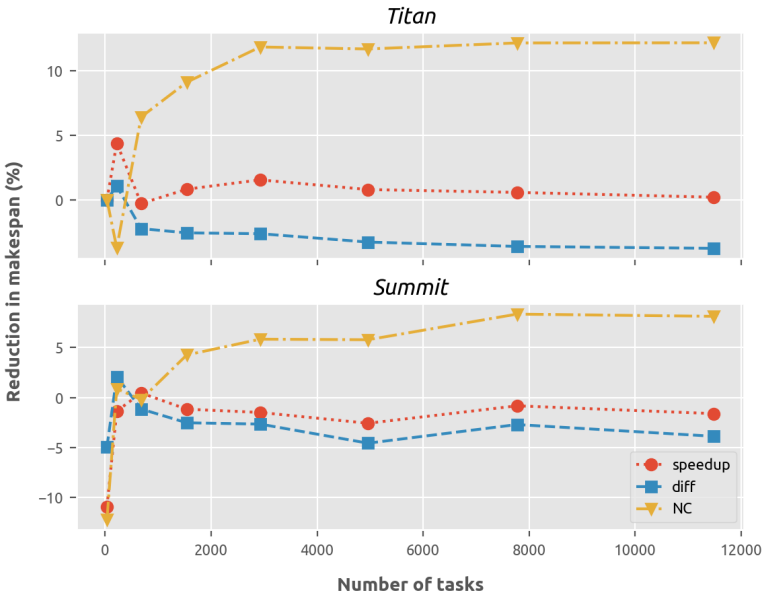

Figure 4.5: Alternative rankings from Shetti, Fahmy and Bretschneider versus default, Cholesky DAGs (tile size 64).

default weighting, particularly as the number of tasks in the DAG grows larger.

To consider the more general case, we repeat the experiments from Table 4.3 for *speedup*, *diff* and *NC*. Table 4.4 below displays the results. As before, it is difficult to draw many clear conclusions about the respective performance of the different rankings in general. The data suggest there are circumstances in which one or more of the alternative rankings outperform the default but the converse is likewise true and certainly none of the alternatives appear to be uniformly superior to the others. These conclusions may seem to contradict those of Shetti, Fahmy and Bretschneider but this is not necessarily the case since the DAGs and environments considered here may differ from those used in their experiments; for example, how we generated task CPU and GPU costs differs from their approach. It should also be noted that even for the Cholesky DAGs (which did more closely match their conclusions) the disparity between different rankings didn't become apparent until the DAGs became fairly large.

As they both performed very well for large Cholesky DAGs, it seems reasonable to compare *NC* and *best*/*simple best* for other large DAGs to establish whether they are capable of improving on the standard HEFT ranking step for more general applications. Preliminary results with larger (5002 tasks) DAGs from the STG however were not encouraging so we eventually decided not to pursue this any further. More extensive testing with large DAGs for these and other alternative task rankings may well be useful in the future.

Another possible task priority list can be derived from what we call the *op-*

Table 4.4: Comparison of HEFT with four different ranking steps for DAGs with 502 tasks from the STG.

	Titan				Summit			
Weighting	APD	WPD	Wins	Fails	APD	WPD	Wins	Fails
Low data								
mean	0.5	3.9	49	0	1.6	7.6	44	0
speedup	0.8	4.3	45	0	3.9	10.2	2	0
diff	0.4	4.8	72	0	1.8	7.9	30	0
NC	2.1	6.1	14	0	0.5	3.3	19	0
High data								
mean	1.8	32.7	97	99	5.1	52.5	80	133
speedup	5.2	27.2	56	115	10.5	49.9	51	133
diff	7.8	44.2	46	118	12.4	57.7	40	129
NC	8.2	48.1	40	127	13.8	55.8	40	135

timistic weighting, the approximation DAG that corresponds to that we use for computing the critical path (see Section 4.1). The basic idea here is that for all tasks we know which processor it should ideally be scheduled on, disregarding contention, so we take the node weights to be the corresponding execution time and compute the edge weights assuming that the tasks are actually scheduled there. Task priorities are then determined by computing the upward rank of all tasks. The immediate issue with this approach is that the optimistic weighting is much more expensive to compute than the default, but here we disregard this consideration in order to investigate only whether there are actually any instances in which it can lead to a significantly reduced schedule makespan at all.

Of course, since we have entirely ignored processor contention there is reason to doubt this model will be useful. However, results for Cholesky DAGs were somewhat encouraging, particularly for large sizes, as can be seen from Figure 4.6, which shows the percentage reduction in makespan we achieve by using the optimistic weighting rather than the default. Note however that these reductions were still not as impressive as those obtained by the considerably cheaper *best* and *NC* rankings. Since further experiments with DAGs from the STG did not suggest any improvement over the default ranking on average, we ultimately concluded that the *optimistic* ranking is not a promising alternative in general.

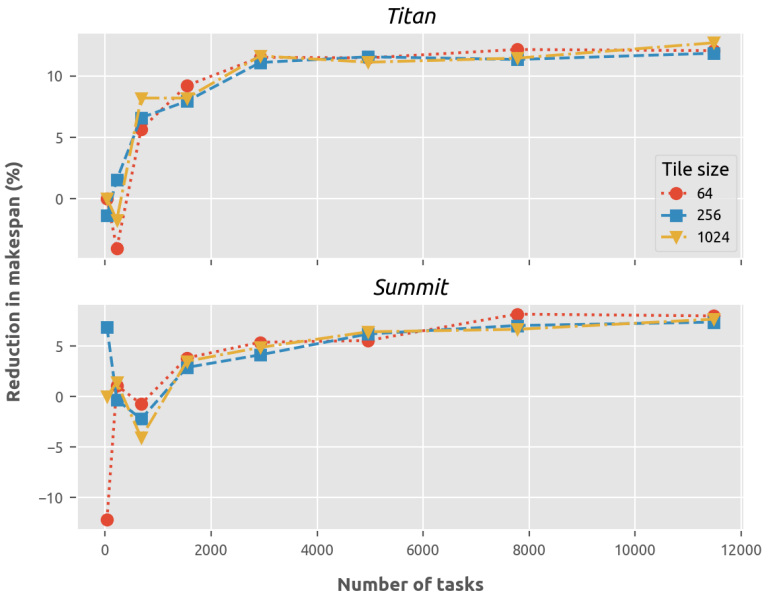

Figure 4.6: Performance of optimistic weighting versus default for Cholesky DAGs.

4.2.3 Alternative processor selection strategies

The standard processor selection procedure in HEFT is for tasks to be scheduled on the processor which is predicted to complete their execution at the earliest possible time. We have already seen examples for which this is problematic because of the myopic nature of this choice (e.g., when we cannot avoid large future communication costs) but in general this seems reasonable and leads to good performance. Perhaps the simplest alternative that is actually sometimes used in practice is to select a processor at random, weighted by their respective speeds. As we would expect, the HEFT processor selection phase improves considerably on this in general; as an example, Figure 4.7 illustrates the reduction in makespan for a selection of Cholesky DAGs. The only really interesting takeaway here is that we again see that the relative improvement declines as the number of tasks in the DAG increases. Precisely why this is the case is unclear but one possibility is that in such situations the optimal distribution of the tasks across the processors resembles how the tasks would be assigned in a weighted random manner.

There are basically two families of alternative processor selection phases that are popular for HEFT (and the other listing heuristics discussed in this chapter). The first is based around the idea of looking ahead to future time steps—i.e., a task's children (or beyond)—when selecting a processor. As described in Section 2.3.3, Bittencourt, Sakellariou and Madeira proposed an extension of this form, which we investigate further in Section 4.5, so we will consider all other

Figure 4.7: Improvement of HEFT processor selection versus random alternative, Cholesky DAGs.

alternatives along these lines there instead.

The other family of alternative task allocation heuristics are based on the idea that tasks have some *preference* for a certain processor or type of processor and we should only schedule the task to another if some criterion is met. An example would be HEFT-NC (see Section 2.3.4) or CPOP (see Section 2.3.2), which aims to schedule all tasks on the critical path on the same processor. Again, as we consider HEFT-NC in Section 4.4, we will leave all investigation of preference-based processor selection phases to there.

4.2.4 Analysis

Taking all of numerical experiments thus far in this chapter into consideration, we make the following comments about HEFT.

- While it often performs well on CPU-GPU platforms, there are also many examples for which it either fails entirely (in the sense that it fails to at least match the minimal serial time) or does not improve on simpler and cheaper heuristics such as RTS. In particular, HEFT appears to struggle when data movement is expensive, because its myopic nature prevents it from incurring large future communication costs at the processor selection phase. More surprisingly perhaps, HEFT was also outperformed by RTS for large Cholesky DAGs and a significant fraction of smaller random DAGs, especially on the Summit node.

- None of the alternative task ranking methods that have been proposed are universally superior to the standard mean value approach, although there are many instances for which at least one of them does achieve a better makespan. For example, the *best* and *NC* rankings were superior for large Cholesky DAGs. However, none of the alternative rankings were capable of compensating for the greediness of the processor selection phase in high-data environments, suggesting that the latter phase is in some sense more critical to achieving good performance on such platforms. Given the mixed results, a more fruitful approach may be a cheap adaptive method capable of deciding which task prioritization step is likely to do well for a given DAG and computing platform. Alternatively, a general method capable of finding a near-optimal schedule given any input priority list of tasks would also be very useful since in practice, it is often the case that users will be familiar with their application and may prefer to set their own task priorities.

- Assuming that a complete task ranking has already been computed, the standard HEFT task allocation step seems to be effective when data movement costs are low but, as mentioned already, can be woefully inadequate when these are high.

4.3 PETS

Although Ilavarasan and Thambidurai evaluate PETS through extensive numerical testing in their original paper, their experimental setup was more general than our own, so we ran a suite of tests to evaluate how well PETS performs for the accelerated environments we are interested in. We use HEFT as a yardstick for comparison because the task ranking step of PETS is essentially an extension of HEFT's and because it is probably the most popular static scheduling heuristic in practice. As stated at the beginning of this chapter, we are not overly concerned with the relative efficiency of the heuristics because the time complexity bounds are well known (see Section 2.3.5) and thus differences may be implementation-specific, but we do note that our own implementation of PETS was about as expensive as HEFT in general.

As noted in Section 2.3.5, there is some flexibility when it comes to computing the DTC since communication costs between tasks are unknown in our framework until they have both been scheduled. Our own implementation calculates the DTC of a task by taking the data transfer cost between a parent and child to be the mean cost over all links, as in HEFT. An alternative that we did investigate is to use the worst-case data transfer cost instead but this never made any significant difference compared to using average costs.

Figure 4.8 shows the reduction in makespan that PETS achieves relative to HEFT for a set of Cholesky DAGs on a single node of both Titan and Summit.

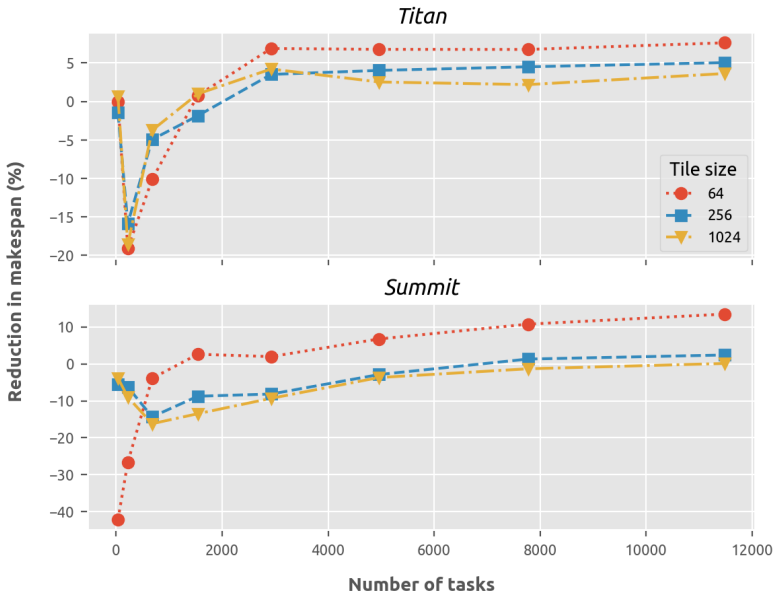

Figure 4.8: PETS improvement over HEFT for Cholesky DAGs.

PETS generally does worse on both platforms for smaller DAGs but improves as they grow larger, being superior for all tile sizes on Titan after the DAG size grows beyond about 1800. One possible explanation for this is that the more complex task ranking of PETS becomes more useful as the number of tasks in the DAG increases, although an alternative is that this is simply another example of HEFT having difficulty with large DAGs. Another interesting takeaway from the figure is that the relative performance of HEFT and PETS seems to depend on tile size to some extent, with the latter being decisively better on Summit for large DAGs when the tile size is 64 but not for the two other sizes considered. (This divergence is perhaps to be expected given that data transfer costs are incorporated as part of the PETS ranking phase.)

Results when we repeated the comparison using DAGs with 1002 tasks from the STG were much less promising for PETS. In the low data case, HEFT was consistently superior for both platforms, resulting in an average makespan reduction of more than 7% in both cases. PETS did perform slightly better under the high data regime, particularly on Summit, recording 19 fewer failures than HEFT on the 180 DAGs considered, but still failed on more than a third of the DAGs and was also the worst of the two on Titan. Since PETS performed best for large Cholesky DAGs, we also considered a randomly selected set of 30 size 5002 DAGs from the STG, however results there were no more encouraging, with HEFT achieving average makespan reductions of around 7% over PETS on both platforms in the low data case.

Our investigation so far suggests that PETS is not well-suited to the accel-

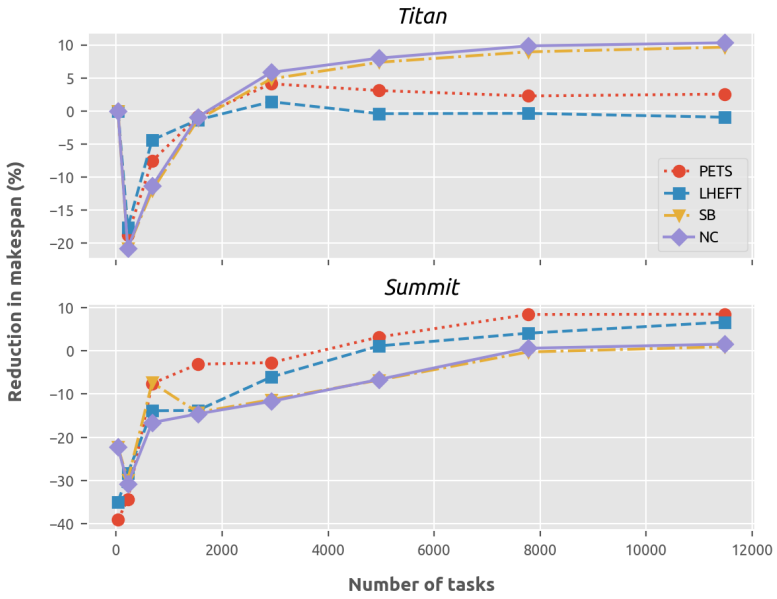

Figure 4.9: Four different task prioritization phases compared with HEFT, Cholesky DAGs (tile size 64).

erated computing environments we are considering. One possible explanation for this is that the level-based approach itself is ill-suited in this case, since the task ranking and processor selection phases of PETS ultimately only apply on a level-basis. To evaluate this, we assume that the levels are defined in the usual way and consider alternative choices for both of the other phases. First, we assumed that the processor selection phase proceeds as normal and considered the following different ways to determine the priorities of tasks in each level:

- Descending order of upward rank, calculated using mean values as in HEFT. (We call this entire heuristic, including the other phases, *Levelized HEFT*.)
- Descending order of upward rank, calculated using best-case values.
- Descending order of Shetti, Fahmy and Bretschneider's modified speedup metric (called *NC* in Section 4.2.1).

Figure 4.9 compares the schedule makespans of these heuristics with HEFT for Cholesky DAGs with tile size 64. Results for other tile sizes that we considered were similar in terms of how the heuristics performed relative to one another, although their performance relative to HEFT could be quite different—for example, for tile size 256, all of them were worse than HEFT for the larger DAGs on Summit. The inconsistency of the heuristics on the two different platforms (e.g., *NC* being probably the best on Titan but the worst on Summit) is perhaps

the most interesting trend we can observe from the plot, but we see again that some of them did outperform HEFT for the larger DAGs. As all of these heuristics, including HEFT, have performed poorly in high data environments so far, we next repeated the comparison for low data DAGs of size 1002 from the STG. These simulations suggested that all three heuristics were consistently worse than HEFT by at least 5% on both platforms, so we ultimately conclude that none of these heuristics is competitive with HEFT in low data environments.

Now we assume that all task priorities have been set as in the standard algorithm and investigate two alternative ways to schedule the (independent) tasks in each level:

- *Min-min*. A classic heuristic for scheduling independent tasks. For all unscheduled tasks in the level, we compute the minimum completion time (i.e., the earliest time at which it can be completed) across all processors. Then, the task with the minimum minimum[1] completion time is selected and scheduled on the corresponding processor. The procedure is then repeated until all tasks in the level have been scheduled. Despite its relative simplicity, an exhaustive comparison study by Braun et al. found that min-min was competitive all other alternatives that they considered [14]. The drawback is that it can be very expensive since the minimum completion times need to be updated every time a task has been scheduled.

- *BMCT*. Heuristic proposed by Sakellariou and Zhao [70], see Section 2.3.6.

Both of these heuristics are more expensive (in our own implementations, up to ten times as long) than the standard PETS processor selection phase but are known to perform well. The idea here then is that if PETS cannot improve on HEFT using these alternatives then it almost certainly will not be able to in general using its default earliest finish time processor selection phase. Figure 4.10 shows the result for Cholesky DAGs with tile size 64; other sizes that we considered were also similar. Although we see that the alternatives are occasionally superior, there is again a lack of consistency in the data, with *min-min* being the best on Titan but the worst on Summit. Further experiments with DAGs from the STG were also not promising. Since most of the failures in the high data case are caused by the greediness of the processor selection phase, it seemed reasonable to hope that the two alternatives may lead to better performance. This was actually true for all the examples we considered but not to a significant degree, with all heuristics still failing on the majority of the DAGs.

Altogether our experiments in this section do seem to suggest that PETS is not well-suited to our target platforms, at least compared to HEFT. This does not necessarily contradict the existing literature since the assumptions of our model are more restrictive than the more general case which is usually considered. For

[1] Hence the name.

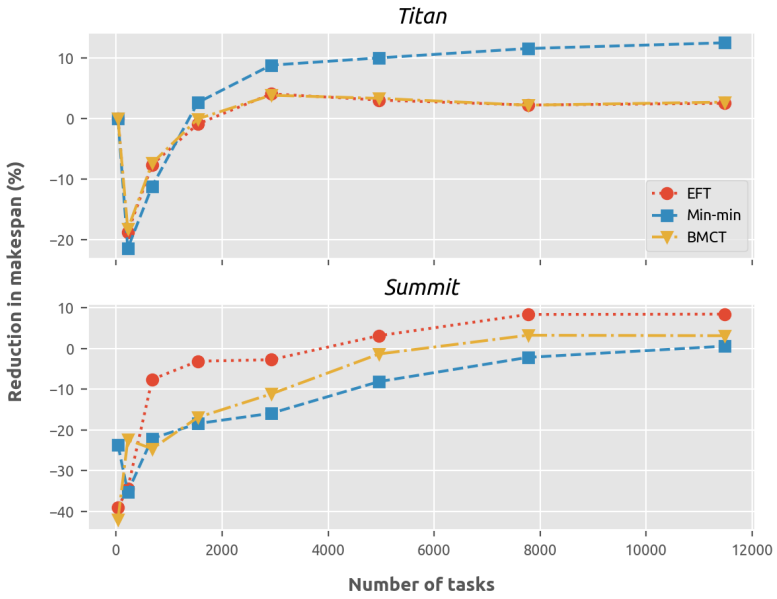

Figure 4.10: Three different processor selection phases in PETS compared with HEFT, Cholesky DAGs (tile size 64).

example, although PETS was inferior to HEFT for the low data DAGs that we considered, when all data movement costs are on the order of the mean GPU time, it is entirely possible that PETS is superior for a wide range of DAGs whose data distribution, while still relatively low, does not follow this pattern.

4.4 HEFT No-Cross

An issue that arose when implementing the HEFT No-Cross heuristic (see Section 2.3.4) is a consequence of the fact that for each task we have only two possible execution times, a CPU time and a GPU time. So in particular this means that there will be multiple fastest processing units (i.e., all the CPUs or all the GPUs) and the original paper does not specify which should be used for the comparison. Our solution is to compute $W_{abstract}$ for all the minimizing processors and pick the one that minimizes the ratio used for comparison with the cross threshold. This seemed reasonable to us, although there are of course alternatives that could be used.

Since HEFT No-Cross was specifically optimized for CPU-GPU environments we would expect that it should improve upon HEFT at the very minimum, so as usual we use that as a comparison. Figure 4.11 shows that this is indeed generally the case for Cholesky DAGs, when we use the value $X = 0.3$ for the cross threshold as suggested by the authors. Results for DAGs from the STG were

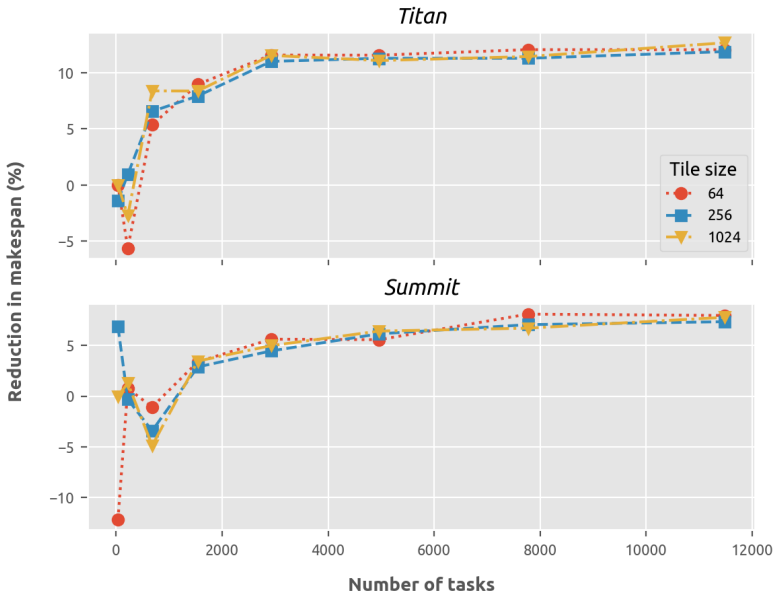

Figure 4.11: HEFT No-Cross makespan reduction compared to standard HEFT, Cholesky DAGs.

more mixed, as we see from Table 4.5. HEFT No-Cross was slightly superior on average on Summit but worse on Titan in the low data case, and consistently worse on both platforms in the high data regime.

It seems clear that HEFT No-Cross finds high data DAGs as problematic as standard HEFT. Indeed, if anything the problem seems more pronounced. This could well be because the modified algorithm is in some sense even more greedy since the crossover is designed to more frequently allow locally optimal (i.e., greedy) actions. Preliminary results suggested that this poor performance cannot be remedied simply by different choices for the cross threshold so we ultimately conclude that the No-Cross heuristic is not well-suited in that case. Of course, in a very real sense the high data DAGs as we define them here are pathological and could be argued to be unrealistic. As with PETS, there may well exist a wide range of data distributions such that HEFT No-Cross performs very well. However what they do help to illustrate is that good performance is very much dependent on how scheduling heuristics account for communication costs. Situations in which something close to the minimal serial policy is optimal may not be very interesting but we would expect that a reliable scheduling heuristic should be able to successfully match this and at least do no worse. Hence for the remainder of this section we focus only on low data DAGs.

Since Figure 4.11 shows that HEFT-NC was clearly superior to HEFT on both platforms for large Cholesky DAGs, we also compared the two for a randomly selected set of 30 DAGs with 5002 tasks from the STG, as shown in Figure 4.12.

Table 4.5: % reduction in makespan of HEFT No-Cross compared with standard HEFT for DAGs from the STG, using suggested cross threshold $X = 0.3$. Here *fails* denotes the increase in the number of failures (/180) (i.e., negative numbers imply No-Cross failed more often).

	Titan				**Summit**			
Data	Worst	Mean	Best	Fails	Worst	Mean	Best	Fails
				502 tasks				
Low	-7.2	-2.6	3.1	0	-4.3	1.0	7.6	0
High	-92.9	-8.1	28.2	−26	-118.7	-14.9	52.5	−2
				1002 tasks				
Low	-7.2	-2.4	4.8	0	-3.6	0.6	7.8	40
High	-40.4	-4.7	23.8	−17	-80.6	-4.2	49.4	−4

Results here were much less impressive, with HEFT being the clear winner on Titan and honors being roughly even on Summit.

It should be noted that since HEFT-NC is identical to HEFT (with the *NC* weighting used to compute task priorities) in the case $X = 1$ that is in some sense a more natural comparison than the standard algorithm. Figure 4.13 illustrates this comparison for two different choices of X. We see that for the smaller value of X, HEFT-NC was nearly always worse than HEFT, and for the larger value it was almost always very close, with only infrequent deviations which were as likely to be worse as better. This was typical of our experience and we were unable to find any values for the cross threshold such that HEFT-NC consistently bettered HEFT with the *NC* ranking phase. Given this, our conclusion is that, for the models we are considering, attempting to restrict the crossover of tasks in this way does not seem to be effective in general.

4.5 HEFT with lookahead

As stated at the beginning of this chapter, we did not intend to consider efficiency at all in this investigation. However, when considering the HEFT-L heuristic of Bittencourt, Sakellariou and Madeira (see Section 2.3.3) it became unavoidable to some extent as the runtime of our simulations was often impractically high, especially for large dense DAGs. In particular, this made comparison with other heuristics, as we do in Section 4.8, difficult. *Edge density* is defined as the ratio of the number of edges of a DAG to the maximum possible for a DAG of that size. For a DAG with n nodes, any given node t can be connected to all nodes but

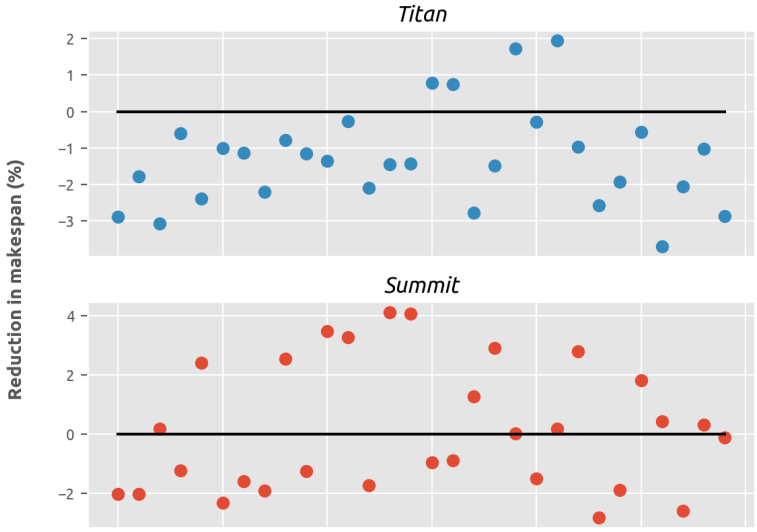

Figure 4.12: HEFT-NC makespan reduction vs. standard HEFT, size 5002 DAGs. The straight line is included to help the reader compare the two overall.

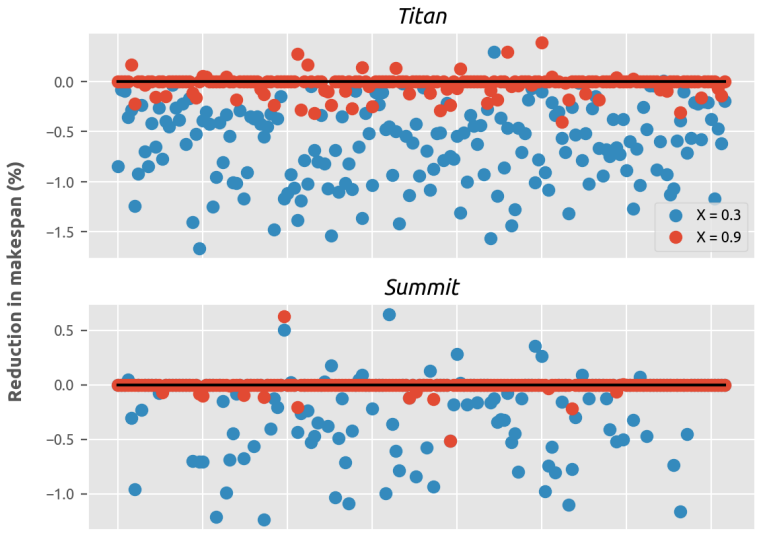

Figure 4.13: HEFT-NC makespan reduction vs. HEFT with *NC* task prioritization, size 1002 DAGs. The straight line is included to help the reader compare the two overall.

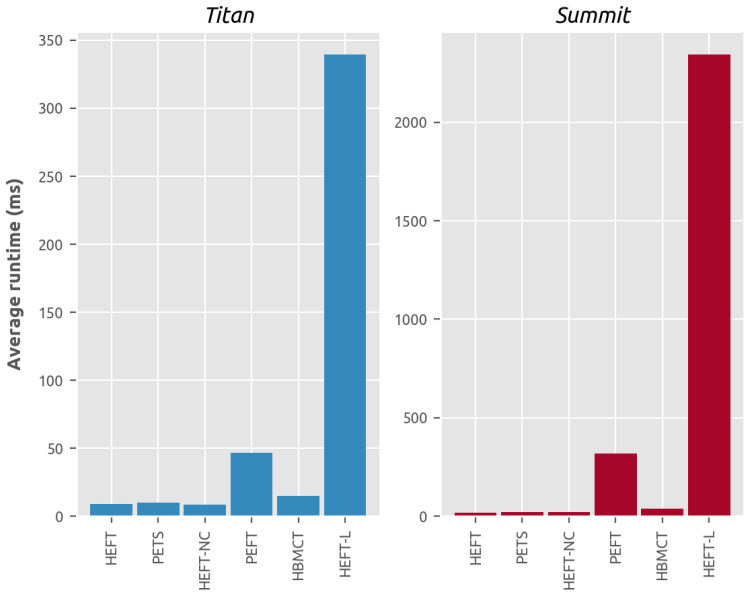

Figure 4.14: Average timings over 10 runs, `rand0046` (52 tasks).

itself (i.e., $n-1$ edges), then the next node we consider can be connected to all nodes but itself and t ($n-2$ edges), and so on. The maximum number of edges is thus $(n-1)+(n-2)+\cdots+1 = \frac{1}{2}n(n-1)$. Cholesky DAGs have a fairly low edge density, usually on the order of 10^{-3} or 10^{-4} (depending on the size) since all tasks have at most two predecessors, which allowed us to compute the lookahead schedule for fairly large DAGs. However for DAGs that were even slightly denser our implementation was unable to handle large examples in a practical time.

For example, `rand_0046` is a DAG with 52 tasks from the STG. It has an edge density of ≈ 0.15 and on average each task has 2.9 parents. Figure 4.14 shows the average time, over ten runs, that our implementation of HEFT-L took to compute a schedule for `rand_0046`. Included for comparison are the timings for all of the other heuristics considered in this chapter. We see that HEFT-L is by far the most expensive. In particular, it was 38 times slower than standard HEFT on Titan and 130 times slower on Summit. Of course, the timings shown are for our own implementations, which are far from optimal. But this disparity will always be expected since the time complexity bounds for HEFT-L are roughly $q\bar{c}$ times higher than HEFT, where q is the number of processing resources and $\bar{c}$ is the average number of children.

Apart from relatively sparse DAGs like Cholesky, time constraints restricted the numerical experiments in this section to DAGs with no more than around 100 tasks. In the immediate future, we intend to use more powerful computational resources than were used here in order to extend this investigation to larger examples.

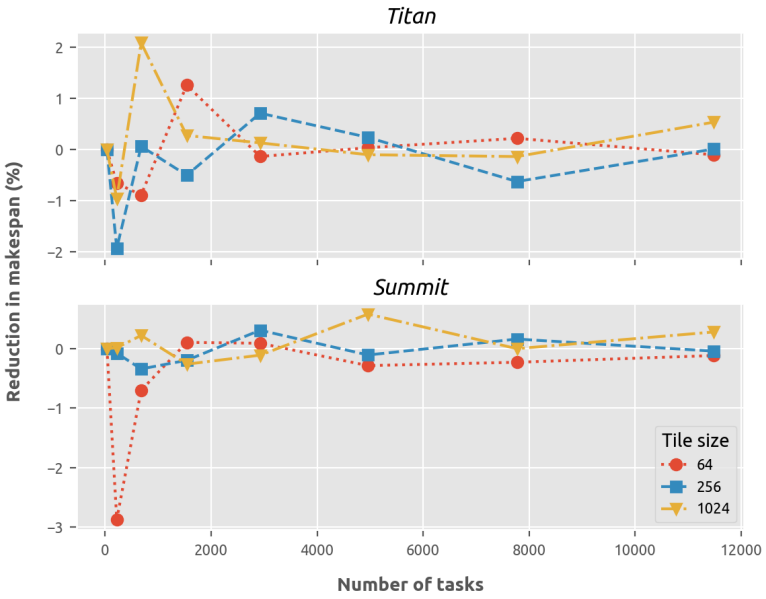

Figure 4.15: HEFT-L compared to standard HEFT, Cholesky DAGs.

Bittencourt, Sakellariou and Madeira considered two different ways to lookahead: selecting a processor that minimizes the maximum (estimated) earliest finish time of a task's children, or choosing the one that minimizes a weighted average of the EFTs of its children. In general we found that there was little difference between the two, although the latter was perhaps slightly more consistent. Hence Figure 4.15 shows the reduction in makespan achieved for the weighted average variant compared to standard HEFT for Cholesky DAGs. We see that there's little difference between the two overall and both perform about as well as HEFT in general.

Table 4.6 compares HEFT-L with the standard algorithm for DAGs from the STG with 52 tasks. Again, the results presented are for the weighted average version as the alternative was very similar. This data is encouraging but two notes of caution are in order. First, the DAGs in question are very small, on the order of the number of processing resources, and thus in some sense pathological. Second, even though HEFT-L always reduced the number of failures and never did any worse for the high data DAGs, its performance considered on its own merits was not that impressive, failing, for example, on 50 of the 180 high data DAGs on Summit. One possible solution would be to lookahead beyond just a task's children, as was considered by the original authors. Unfortunately this will of course be even more expensive, perhaps impractically so for very large DAGs and computing platforms with lots of processing resources.

In an attempt to use the concept of lookahead but also reduce its cost, we considered a simple modification that only looks at one child of each task instead

Table 4.6: % reduction in makespan for HEFT-L compared to standard HEFT for DAGs with 52 tasks from the STG. Here *fails* denotes the increase in the number of failures (/180) (i.e., negative numbers imply HEFT-L failed more often).

	Titan				**Summit**			
Data	Worst	Mean	Best	Fails	Worst	Mean	Best	Fails
Low	-3.8	1.9	34.0	+12	-4.7	0.7	12.3	0
High	0.0	1.8	82.0	+5	0.0	5.5	74.8	+13

Table 4.7: % reduction in makespan for HEFT-L with *random* child sampling compared to standard HEFT, DAGs with 102 tasks from the STG. Here *fails* denotes the increase in the number of failures (/180) (i.e., negative numbers imply more failures than HEFT).

	Titan				**Summit**			
Data	Worst	Mean	Best	Fails	Worst	Mean	Best	Fails
Low	-4.3	-0.3	1.2	+4	-23.1	-3.2	7.6	0
High	-7.9	3.0	62.0	+13	-4.1	8.3	79.9	+22

of them all. The idea here is that in the kind of accelerated environments that we are considering, when we select a processor we aren't so much searching for the one which minimizes the finish times of all its children so much as we are looking for one that is likely to avoid incurring any large communication penalties, which should be apparent when considering any of its children. We investigated two different ways to choose the child: entirely at random or based on which had the highest rank. Both performed almost identically so we used the random version for all results presented in the rest of this section. For Cholesky DAGs, we found that the child sampling variant performed about as well as HEFT-L overall, and Table 4.7 compares its performance with HEFT for DAGs from the STG. Pleasingly, we see that the performance for high data DAGs does improve on HEFT—but it still failed on 70 of the 180 high data DAGs on Summit, which is obviously undesirable.

Unfortunately since the cost of the lookahead scales with the number of processing units, the algorithm can still be expensive even when the number of children considered is only one. In an attempt to further minimize costs, we considered a modification which doesn't attempt to estimate where the child task will be executed but rather computes the finish time on a randomly chosen subset of

Table 4.8: PEFT task prioritization information for DAG from Figure 4.16.

Task	OCT(CPU)	OCT(GPU)	Rank	Priority
1	567.0	205.4	386.2	2
2	315.2	79.1	197.1	3
3	782.2	79.1	430.6	1
4	0	0	0	4

the processors. The idea here again being that this may still be useful by avoiding large immediate communication costs. Comparisons with HEFT-L for Cholesky DAGs suggested this approach was competitive with the full lookahead heuristic even when we consider only one randomly chosen CPU core and one random GPU, although results were much more mixed for DAGs from the STG (with 502 tasks). On Titan, the processor-sampling lookahead heuristic did about as well as HEFT-L, bettering standard HEFT on high data DAGs and being competitive for the low data ones. But on Summit it was an abject failure in the high data case, increasing the makespan by a factor of four on average, and about 7% worse on average for low data DAGs. Even extending the sampling to one CPU core and all of the (six) GPUs did not improve performance. Still, it seems safe to say that in some situations at least randomized sampling is just as effective as doing a full lookahead step, and this is something that we may investigate in greater depth in the future.

4.6 PEFT

As mentioned in Section 2.3.7, the suggested task prioritization phase in the PEFT heuristic can sometimes lead to invalid task priorities for the target platforms we are considering. For example, consider the following simple DAG which is to be scheduled on a simple heterogeneous platform with one CPU and one GPU, connected by a Gemini interconnect (bandwidth 8912 B/μs).

Table 4.8 summarizes the relevant ranking information (timings in μs). We see that Task 3 is assigned a higher priority than its parent Task 1, clearly violating precedence constraints. As already noted, a simple solution to this issue is to instead use the *minimum* OCT value over all processors instead, which is what we do by default in our implementation.

Figure 4.17 shows the reduction in schedule makespan that PEFT achieves relative to HEFT for Cholesky DAGS on single nodes of Titan and Summit. In general, PEFT is the better, particularly for larger DAGs, although there are a significant number of the DAGs for which this is not the case. Results for DAGs

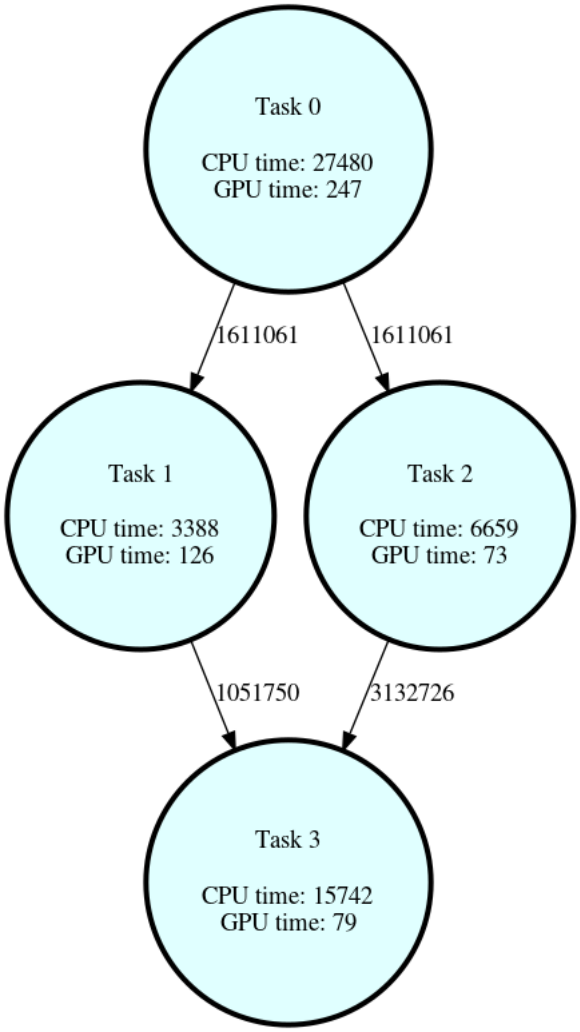

Figure 4.16: Simple DAG for which the mean OCT value prioritization phase in PEFT breaks down. Task execution times are in μs and rounded to the nearest integer for clarity. Edge weights are data transfer amounts in bytes.

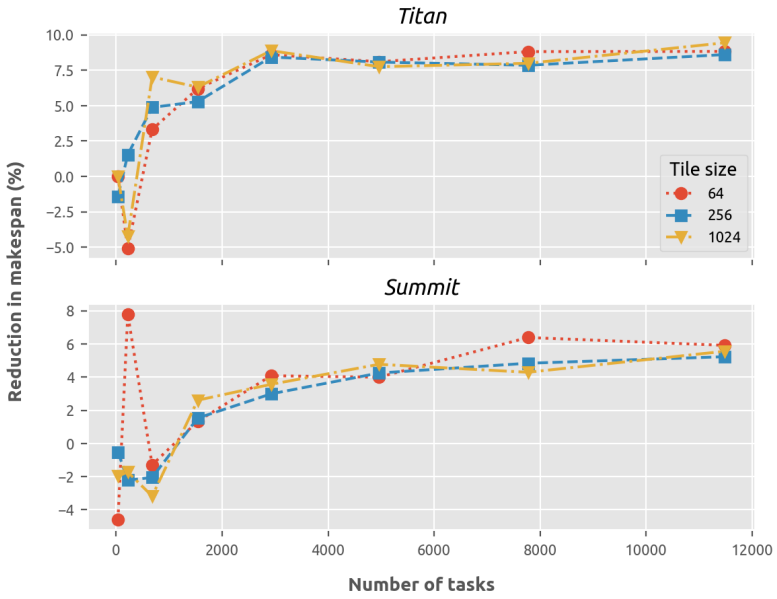

Figure 4.17: PEFT makespan reduction vs. HEFT, Cholesky DAGs.

from the STG were similarly mixed. Table 4.9 compares the PEFT schedule makespan to that computed by HEFT for DAGs with 502 tasks, for both high and low data distributions. HEFT achieves a smaller makespan for the low data DAGs on both platforms, while PEFT is slightly superior in the high data case, with significantly fewer failures (although the raw numbers are still high) for Titan but considerably worse on the Summit node.

There were several possible modifications that we investigated for PEFT. For example, we could use any other task prioritization phase in the algorithm. Figure 4.18 compares standard PEFT and a variant called PEFT-NC that uses the *NC* task ranking phase with HEFT for Cholesky DAGs. While the modified version

Table 4.9: % reduction in makespan for PEFT versus HEFT, DAGs with 502 tasks from the STG. Here *fails* denotes the increase in the number of failures (/180) (i.e., negative numbers imply more failures than HEFT).

	Titan				**Summit**			
Data	Worst	Mean	Best	Fails	Worst	Mean	Best	Fails
Low	-9.8	-3.9	0.9	0	-6.5	-0.2	5.7	0
High	-25.3	1.5	41.2	+37	-146.3	-11.1	28.4	+2

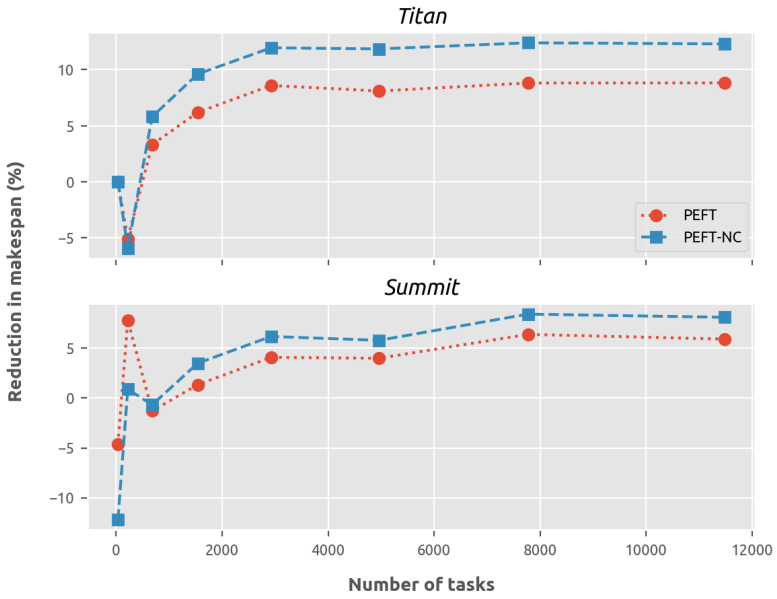

Figure 4.18: PEFT-NC makespan reduction vs. HEFT, Cholesky DAGs (tile size 64).

is usually superior to both HEFT and standard PEFT, it was not a significant improvement over, for example, HEFT with the *NC* weighting, which is cheaper. This was typically the case for all of the possible optimizations that we considered and we were unable to find any that were superior to cheaper alternatives.

4.7 HBMCT

Originally proposed in the context of general heterogeneous platforms, one minor issue that arose when implementing HBMCT was how we should initially assign the tasks to the processing units in the BMCT phase: since all tasks have only two possible execution costs in our model (i.e., a CPU cost and a GPU cost), there may be multiple "fastest" processors. Our solution was to randomly select one from this set. Figure 4.19 compares HMBCT with HEFT for Cholesky DAGs of various sizes. We see that HEFT is consistently the best of the two. Results for DAGs with 1002 tasks from the STG were even less encouraging. In the low data regime, HBMCT was on average 10% worse on Summit and just under 1% worse on the Titan node. In the high data regime, HBMCT was on average about 70% worse on both platforms.

While different initial task allocations do affect the overall performance, we were unable to determine any choices that led to consistently better performance than HEFT. For example, by assigning all tasks in each group to the processor

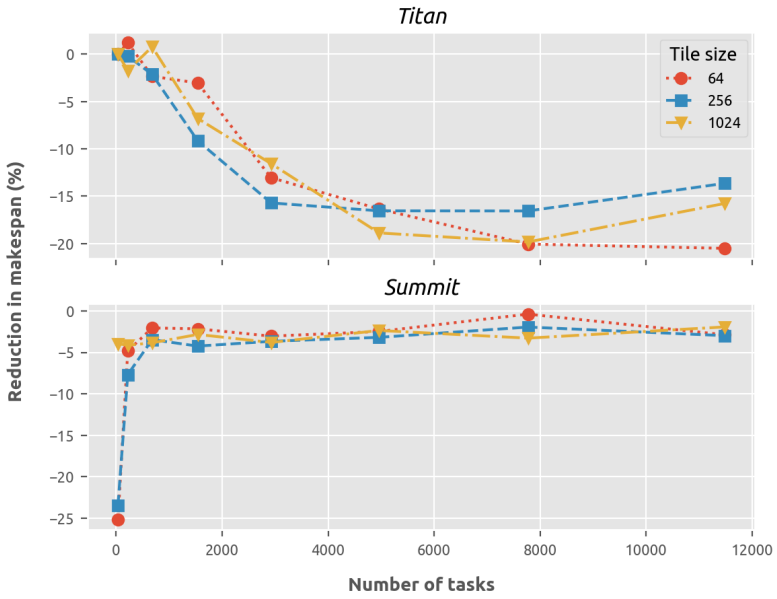

Figure 4.19: HBMCT makespan reduction vs. HEFT, Cholesky DAGs.

that minimizes its earliest finish time (as in HEFT), HBMCT became slightly superior on average on Titan and slightly worse on Summit in the low data case. The relative performance on Summit in the high data case also improved slightly, although was still worse than HEFT (which itself does very badly) by about 50% on average. These results are of particular concern since HBMCT with EFT task allocation is just HEFT with the addition of the BMCT step, suggesting that it may actually be detrimental in this case. This is somewhat surprising considering that by construction BMCT necessarily does not increase the current overall makespan after the scheduling of each group. The most plausible explanation we suggest is that this is another example of local greediness leading to adverse global consequences. As noted in Section 2.3.6, HBMCT can be considered an analogue of HEFT that operates at a group level, so improving the quality of the group scheduling may inadvertently lead to future penalties (either by incurring extra communication costs or obstructing the scheduling of later tasks) in a similar way as can happen with HEFT.

In an attempt to determine if including the grouping step ever leads to consistently superior performance over HEFT, we considered the following simple alternative ways to schedule the tasks in each group.

- *Minimum execution time* (MET). A classic scheduling heuristic. All tasks are scheduled on the processor with the smallest execution time. Note that this is the default initial task allocation phase in HBMCT so this comparison will allow us to evaluate the effect of the BMCT phase.

- *Min-min.* Described in Section 4.3.
- *Max-min.* Very similar to above but after computing the minimum completion times for all unscheduled tasks over all processors, the task with the *maximum* overall completion time is selected and scheduled first.

Comparing the schedule makespans obtained by HBMCT using these alternative group scheduling policies to the BMCT default (and also HEFT as a baseline) for DAGs with 502 tasks from the STG on simulated nodes of Titan and Summit, we found the following:

- BMCT consistently improved performance compared to MET in the low data regime, on average by about 10% on Titan and 30% on Summit. But it was still worse than HEFT on average in both cases, slightly on Titan (0.6%) and significantly on Summit (12.1%). (Note that here we are using BMCT with initial task allocation as in MET, not the EFT alternative discussed above.)
- Results were more mixed for the high data DAGs. BMCT improved on MET for Summit, while still performing very badly overall (143/180 failures), but was detrimental on Titan, leading to 57 more failures and reducing the mean makespan reduction compared to HEFT by more than 70%. In fact, MET was better than all of the alternatives, including HEFT, for high data DAGs on Titan.
- Min-min and max-min were slightly more capable than BMCT in the high data regime, especially on Titan where both reduced the raw number of failures by more than 25, but still did badly overall.
- In the low data case both performed better than BMCT and about as well as HEFT. There was no clear winner between the two: min-min was slightly superior on Titan but max-min was on Summit.

Overall, our conclusion from these numerical experiments is that while we cannot say that the grouping step is necessarily detrimental on the accelerated platforms we are interested in, there are situations in which it can lead to significant makespan increases, particularly when communication costs are high. An approach perhaps worth investigating in the future is whether some form of lookahead can be incorporated at the group level, although it is not obvious how this should be done.

So far we have only considered HBMCT with the default task prioritization phase, upward rank with mean value weighting (as in HEFT), but any other is possible. Since upward ranking with *NC* weighting has consistently performed well throughout the experiments detailed in this chapter that seems a reasonable choice for comparison. Figure 4.20 illustrates the results on Titan and Summit

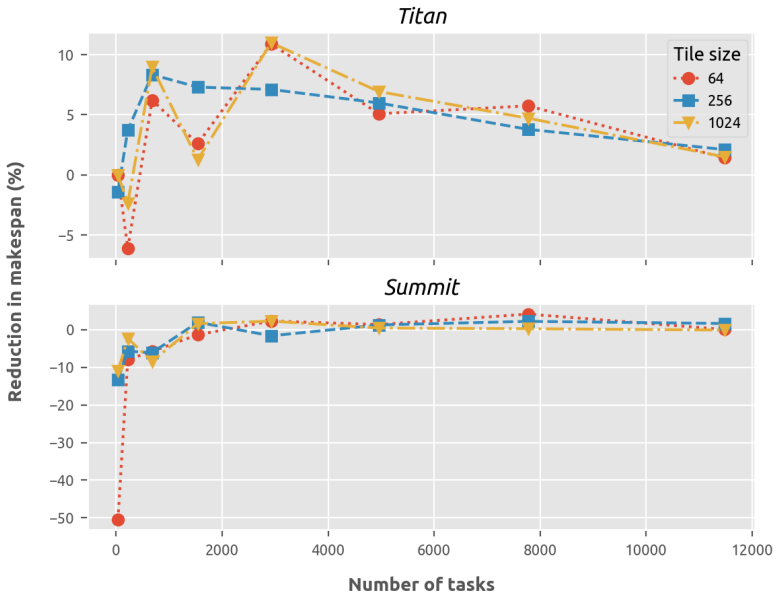

Figure 4.20: HBMCT with NC task priorities vs. HEFT, Cholesky DAGs.

nodes for Cholesky DAGs. Comparing this figure with Figure 4.19, it certainly appears that the modified heuristic is more competitive and at least occasionally superior, unlike the original. However, repeating this comparison for DAGs with 1002 tasks from the STG on the Titan and Summit nodes, there was little difference between the two variants, and both were always inferior to HEFT.

In the context of HBMCT, the intuitive goal when dividing tasks of a DAG into independent groups for scheduling is maximizing group size, the idea being that reducing the number of smaller subproblems to solve will reduce the scope for conflict between balancing optimal local (group-level) and global task scheduling. A similar issue frequently arises in *just-in-time* scheduling, where the goal is typically to find an *IC-optimal* schedule such that the set of tasks ready for scheduling is maximal at every time step [53]. While such a schedule can be shown to exist for many classes of structured DAGs, in general it may not. However, if we relax the goal to finding the schedule that maximizes the number of ready tasks *on average* then this can be done for all DAGs; this is known as *AREA-oriented* or *AO* scheduling [24]. Algorithms exist for computing AO-optimal (or IC-optimal when possible) schedules, although these tend to be complex and inefficient. A more practical priority-based heuristic for maximizing ready task sets has also been proposed by Zhang, Tang and Sakellariou [109]. It may be useful in the future to consider how these ideas can be adapted to generating maximal groups for HBMCT and whether this improves performance.

4.8 Comparisons and analysis

In this section, we directly compare the performance of the listing heuristics we have discussed in this chapter, with the exception of HEFT with lookahead because of lengthy runtimes. In the future we intend to repeat this comparison, or similar, with HEFT-L using more powerful computational resources that will permit this.

Since it seems clear that all of the heuristics are incapable of handling high data DAGs for which all (nonzero) communication costs are roughly the same size as the average CPU task execution time, for the rest of this section when we refer to DAGs from the STG we mean the low data versions. Although PETS, HEFT-L and PEFT all generally proved to be superior to HEFT for these DAGs, their performances were still poor overall, failing on roughly a third of the DAGs even in the very best cases that we observed. Since the issue here is an inability to avoid taking locally optimal actions that lead to large future communication penalties, looking ahead past a task's immediate children would seem to be the promising way to dealing with this. However, this is typically very expensive. A potentially cheaper solution would be task duplication (see Section 2.2.2), which we do not consider in this report but may do so in future research.

Figure 4.21 directly compares the speedup of HEFT, PETS, HEFT-NC, PEFT and HBMCT, as well as HEFT with the *NC* task prioritization (which is labeled HEFT (NC) here) for Cholesky DAGs with tile size 64. We see slightly different relative performance depending on the DAG size and the platform, but overall HEFT No-Cross and HEFT (NC) appear to be the best.

Results were even more ambiguous for DAGs from the STG. Figure 4.22 shows the average percentage degradation (see Section 4.2.2) for the same heuristics for DAGs with 502 tasks from the STG. Some trends are apparent however. First, HEFT and its derivatives were probably the best performing heuristics overall, being the most consistent and always within the top four on both platforms. In contrast, PETS was clearly the worst, having the highest APD on Titan and second highest on Summit. HBMCT had the most mixed performance as it was second best on Titan but the worst on Summit.

Of course, throughout this chapter we have often observed different behavior depending on the DAG size. Since we see from Figure 4.21 that the relative performances of the heuristics became fixed after the number of tasks reached around 5000, we also compared their APD for 30 randomly chosen DAGs with 5002 tasks from the STG, as shown in Figure 4.23. Unfortunately, PEFT was not included in this comparison because of overly long runtimes. Smaller-scale testing with DAGs of this size suggested it was competitive with the best of the alternatives, at least on the simulated Titan node, so we intend to repeat this comparison when possible. Again we see that HEFT and its two variants are the best performing heuristics, especially on the Summit node.

Overall it seems safe to conclude that, assuming data movement costs are

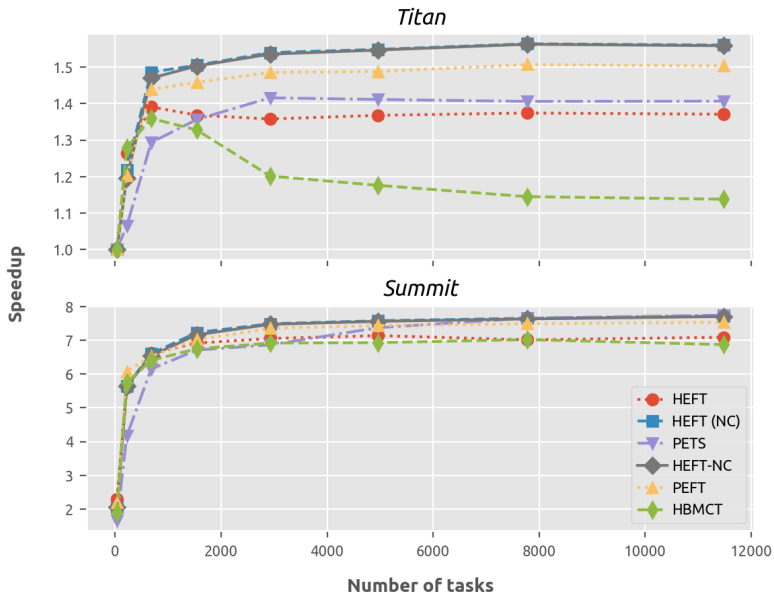

Figure 4.21: Speedup comparison, Cholesky DAGs (tile size 64).

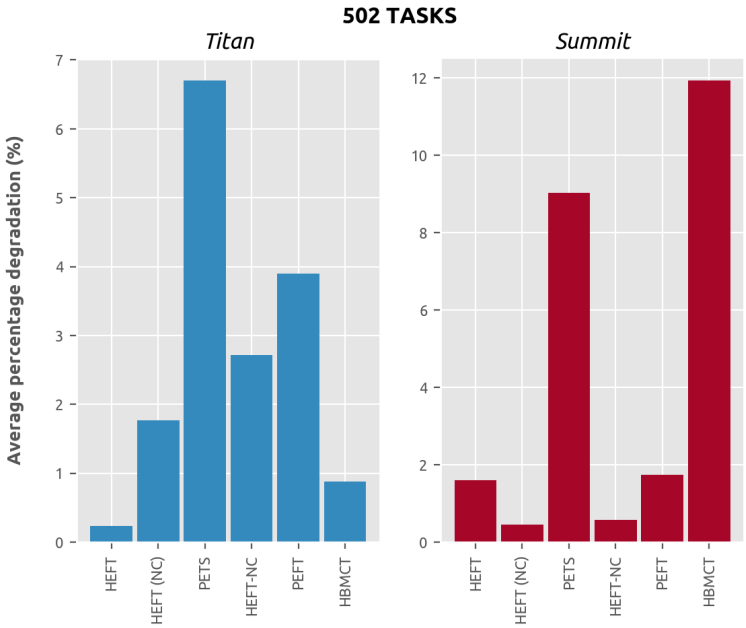

Figure 4.22: Average percentage degradation of listing heuristics, 180 size 502 DAGs from the STG.

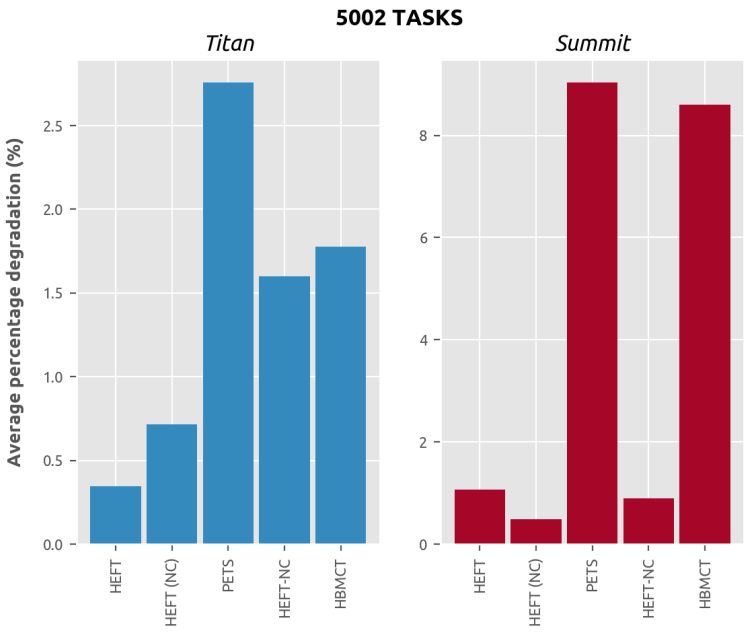

Figure 4.23: Average percentage degradation of listing heuristics, 30 size 5002 DAGs from the STG.

low, which of the listing heuristics performs best depends to a large extent on the target platform and the size (and possibly topology) of the application task DAG, but the classic HEFT algorithm was perhaps the most consistent overall, especially when we allow the use of alternative task prioritization phases (such as *NC*). However, a more exhaustive comparative study—including PEFT and HEFT-L for large DAGs at least—may be useful in the future.

4.8.1 Robustness

Although, as per Section 1.4, we are focused on the deterministic task scheduling problem in this report, in this section we briefly consider the robustness of the static scheduling heuristics discussed in this chapter. This is not intended to be a rigorous investigation but merely an attempt to ascertain, at a high level, whether there are any very prominent differences in this regard between them.

First, we compared the initial schedule makespan computed with the schedule makespan obtained when all task execution costs, on CPU and GPU, were changed to a random value of the same order of magnitude, for Cholesky DAGs. Data transfer amounts and bandwidth values were not altered so communication costs remained identical. Figure 4.24 shows the percentage change in *speedup* of the modified schedule compared to the original on a single node of both Titan and Summit for HEFT, PETS, HEFT No-Cross, PEFT and HBMCT. The tile size is 64 but results were similar for other sizes we considered. HEFT with lookahead is

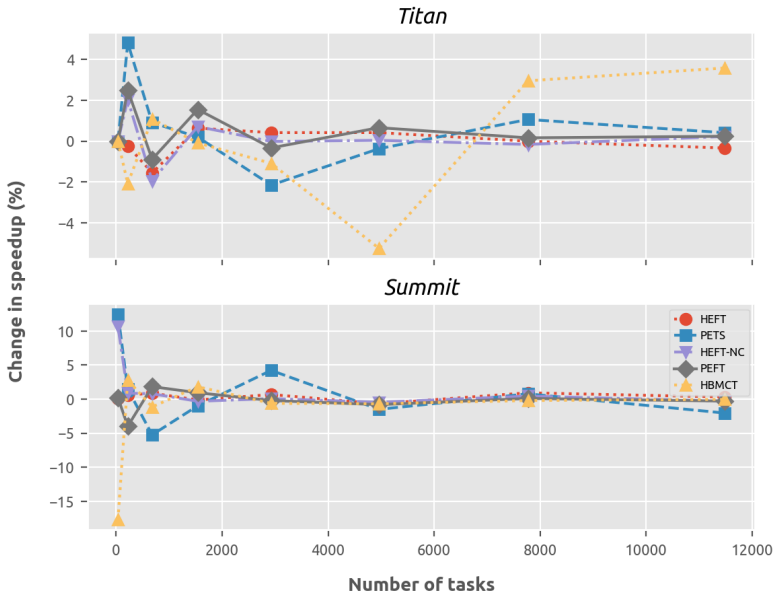

Figure 4.24: Robustness of listing heuristics, Cholesky DAGs (tile size 64).

not shown here due to lengthy runtimes for the larger DAGs proving impractical but testing with smaller DAGs suggested that results were similar to the other heuristics. We see that while the change in speedup seems small in general, this is not always the case, especially on Summit. Note also that even minor fluctuations in speedup can still be significant, especially when the minimal serial time is close to optimal.

For DAGs from the STG, we also modified the amount of data to be transferred in the same way to the execution times (i.e., randomly chosen but still the same order of magnitude)[2]. Table 4.10 summarizes the changes in speedup for HEFT, PETS, HEFT-NC and HBMCT. HEFT with lookahead is omitted due to impractical runtimes, as is PEFT also this time, but again their behavior for smaller DAGs appears to be similar to the others. In general, all four heuristics seem to do fairly well, averaging roughly 1% variation on both platforms. Worst case values were also fairly small, albeit possibly still significant depending on the actual numerical value of the speedup. Overall, there appears to be little difference between the four heuristics. Their relative performance also seems to be platform-dependent to an extent, since, for example, HEFT is probably the most robust on Summit but the least on Titan.

From these and other numerical experiments, we concluded that none of the

[2]In general, these are likely to be known more accurately (certainly in the case of NLA applications) but the actual data transfer times themselves aren't due to uncertainty around the effective bandwidth. We chose to do it this way as it was simpler programmatically but this is just a means to the end of modifying the communication time.

Table 4.10: % change (absolute value) in speedup, 180 DAGs with 1002 tasks from the STG.

	Titan		**Summit**	
Heuristic	Mean	Worst	Mean	Worst
HEFT	1.1	5.5	1.4	5.8
PETS	0.9	4.8	1.6	7.4
HEFT-NC	1.0	4.7	1.4	7.0
HBMCT	1.2	4.4	1.5	6.5

six listing heuristics considered in this chapter (HEFT, HEFT-NC, HEFT-L, PETS, PEFT, HBMCT) are clearly and unambiguously either superior or inferior to any of the others in terms of robustness, although this does not preclude the possibility that any of them are consistently so to a less noticeable degree; we may undertake a more thorough comparative study in future to determine if this is indeed the case.

Note that here we have not considered whether the heuristics are actually sufficiently robust to be useful in practice, as we also intend to investigate this in future work targeting stochastic DAG scheduling. Based on the data presented in this section and our other small-scale experiments thus far, this is difficult to evaluate. Given that the modifications we made to the DAG weights in our experiments were potentially quite severe, being in all cases merely the same order of magnitude, we were somewhat surprised by how robust the heuristics appeared to be. However, as already noted, our investigation was very limited and we will consider this much more rigorously in later research.

4.9 Conclusions

We draw the following conclusions from the numerical experiments detailed in this chapter as a whole.

1. It is clear that the performance of all of the listing scheduling heuristics investigated is highly dependent on the amount of data movement in the DAG, with none able to adequately cope with the (perhaps unrealistic) case of all (nonzero) communication costs being on the order of the average CPU task execution time, although further research is intended for HEFT with lookahead in particular. Task duplication is an alternative that we also intend to consider in the future.

2. When data movement costs are low, most of the heuristics performed well, based on metrics such as SLR and speedup. Determining which is the most suitable for a given DAG and target computing platform is difficult, as their relative performances vary according to DAG size, ratio of CPU and GPU resources (and their speeds), and other factors. However, the relatively simple HEFT was perhaps the best performing in general, especially if we consider the use of alternative task prioritization phases.

3. While we did not intend to consider the computational cost of the heuristics themselves, it often became unavoidable and lookahead heuristics in particular were often impractically expensive. This is perhaps to be expected given that their cost typically scales with the number of processing units and tasks in the DAG but, to an extent, did hinder our ability to fairly compare their performance with alternatives for DAGs of practical sizes. Alternative sampling-based approaches may improve efficiency but can be problematic, especially for multiple-GPU target architectures; further research is intended here.

4. The effect of uncertainty with regards to computation and communication time estimates needs to be considered in greater detail. Again, this will be the subject of a future study.

5. Some of the assumptions we made were arguably unrealistic, such as completely neglecting CPU-CPU data transfer times and estimating CPU and GPU execution times based only on their theoretical peak performance; it may be useful to repeat these comparisons without them in the future. Similarly, evaluating the reliability of the simulated results using real data would also be very useful.

Chapter 5

Reinforcement learning

Many scheduling heuristics, including several that we have discussed in this report, are criticality-aware, based on the idea of planning along the critical path of the DAG. If our knowledge of the system is complete and perfect, finding the critical path of a DAG could of course be solved through *dynamic programming*. However the full problem is typically so large (or stochastic) as to be intractable for classical DP methods. However it may be solvable by *approximate dynamic programming*, also known as *reinforcement learning* (RL).

Reinforcement learning is a kind of machine learning which can be considered either as a specific form of supervised learning or as a distinct type of learning in its own right [81, p. 3]. The key idea is learning through *interaction*. The framework for a reinforcement learning problem is as follows: an *agent* interacts with an *environment* whose dynamics are not entirely known. The agent observes the *state* of the environment and then takes some *action* based on this information for which it receives feedback in the form of some scalar *cost*. The goal of the agent is to minimize the total cost that the agent incurs over all time. (Note that the aim of reinforcement learning is often framed in terms of maximizing some *reward* rather than minimizing costs, but the two are equivalent and we prefer this formulation as it is more intuitive for the task scheduling problem, where the aim is to minimize schedule makespan.)

Reinforcement learning as a concept has been around for decades but was traditionally impractical for interesting problems because it was unable to deal with the large number of possible states and actions that these problems usually possess; this was the so-called *curse of dimensionality*. However some results in recent years, particularly in *deep* reinforcement learning, suggest that this may no longer be the case. In particular, the successes achieved by Google's DeepMind in training deep reinforcement learning agents to play Atari games at superhuman levels of performance [57] and defeat the best human players at the board game Go [74] clearly show that reinforcement learning is now a viable method for dealing with complex problems. Due at least in part to these achievements, reinforcement learning has enjoyed a surge in popularity in the last few years and

has been applied to a wide range of problems in many application areas.

5.1 Background

In this section we briefly summarize some of the key concepts from reinforcement learning that we will need going forward.

5.1.1 Markov decision processes

More formally, we consider the interaction of an agent and an environment over a discrete series of time steps, $t = 0, 1, 2, \ldots$, although extensions to the continuous case are possible (see e.g., [8]). At each time step the agent observes the state of the environment, $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states, and selects an action $a_t \in \mathcal{A}(s_t)$, where $\mathcal{A}(s_t)$ is the set of all actions that the agent may take when the environment is in state s_t. After taking action a_t, the agent incurs some cost c_{t+1} at the start of the next time step, when it also moves into a new state s_{t+1}. When applying this framework to a problem, how we define the state of the environment and the actions that we may take is clearly vital but, ultimately, actions can be any decision that we want to learn how to make and states anything we need to know in order to decide how to make them. Similarly, time steps do not have to represent literal time. They could instead represent stages of a project, for example, or any other useful discretization of the decision problem that we are trying to solve.

The goal of the agent is to minimize the total cumulative cost that it incurs over all time. If the problem is *episodic*—guaranteed to end within some finite number of steps—then we can simply sum all of the costs incurred at every time step but if the problem is infinite then this sum may not converge. In addition, it often makes sense intuitively to weight present costs more heavily than future ones. The usual way to handle this is to introduce a *discount factor* $\gamma \in [0, 1]$ such that the agent takes actions to minimize the sum of discounted cumulative costs. Let C_t be the *expected* total future cost after time step t. Then the agent actually seeks to minimize

$$C_t \coloneqq c_t + \gamma c_{t+1} + \gamma^2 c_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k c_{t+k}.$$

Clearly, if $\gamma = 0$ then we will simply choose the action that minimizes the immediate cost at every step and as γ approaches one we take future costs more heavily into account. Introducing a discount factor can often make problems more tractable analytically and is very common in practice.

We should like to define the state of our environment in such a way that it summarizes all relevant information about previous states and tells us everything

we need to know in order to choose the action we take. Such a system in which the current state encapsulates all the information we need in order to make a decision is called a *Markov decision process* (MDP). More formally, this means that if the environment is in a state $s_t = s$ then the probability that the next state is $s' = s_{t+1}$ is independent of the history of the process so far. Sutton and Barto note that while this framework may not be adequate for describing all decision problems accurately it has nonetheless proven to be very useful [81, p. 49].

5.1.2 Policies and value functions

A *policy* π defines how an agent behaves. Essentially, it is a map from states to the actions the agent may take when it is in that state. In general, policies may be either deterministic or stochastic. A policy is called *stationary* if the distribution of probabilities of taking an action when in a given state depend only on that state—and so, in particular, if the policy is deterministic, the agent always takes the same action whenever the system is in that state. The *optimal* policy π^* is the one which minimizes the total cost we incur over all time. It can be shown that, under certain reasonable assumptions, at least one optimal policy always exists [8] and we will generally assume that this is the case.

We attempt to balance immediate costs against potential long-term savings through the *value* or *cost-to-go* function, which essentially tells us how good it is in the long-run to be in a certain state. Typically the value of a given state will be the total cumulative cost that the agent can expect to incur starting from that state. This doesn't necessarily have to be the case but the important thing is that the value function encodes the relative long-term differences in costs that we can expect to incur from different states. We will generally use V to denote value functions and V_π to denote the value function when we operate under the policy π (so V_π(s) is the value of being in state s when following the policy π).

More formally, we define the value of a state $s_t = s$ under a policy π to be the expected total cost after starting in state s and continuing with policy π thereafter, i.e.,

$$\begin{aligned} V_\pi(s) &\coloneqq \mathbb{E}_\pi[C_t \mid s_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k c_{t+k} \;\middle|\; s_t = s\right], \end{aligned}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π. Here we have included a discount factor γ; for a non-discounted problem, we can simply take this to be one.

It is sometimes useful to define a related function that represents the value of taking particular actions in a given state. For a state s, if we take action a under policy π then we define the *action-value function* (sometimes called the

Q-function) under policy π by

$$\begin{aligned} Q_\pi(s,a) &\coloneqq \mathbb{E}_\pi[C_t \mid s_t = s, a_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k c_{t+k} \;\middle|\; s_t = s, a_t = a\right]. \end{aligned}$$

We typically denote the optimal value function—i.e., the value function of the optimal policy π^*—by $V_{\pi^*} = V^*$ and the optimal action-value function by $Q(s,a)$. Again, there is a lot of analysis concerning this but in general it is fairly safe to assume that these optimal functions actually exist.

It can be shown that the optimal value function must obey the *Bellman equation*

$$\begin{aligned} V^*(s_t) &= \min_\pi \big\{V_\pi(s_t)\big\} \\ &= \min_{a_t} \big\{c_t + \gamma \mathbb{E}[V^*(s_{t+1})]\big\}, \end{aligned}$$

or, in terms of the action-value function,

$$Q^*(s_t, a_t) = \mathbb{E}\left[c_t + \gamma \min_{a_{t+1}}\{Q^*(s_{t+1}, a_{t+1})\}\right].$$

5.1.3 Algorithms

The Bellman equation is the basis for many popular reinforcement learning algorithms such as *Sarsa* and *Q-learning*. At a high level, both of these work by repeating the following process for a given number of *episodes*: start with a policy, evaluate it, then improve it. The latter is done by updating the value function estimates for states and actions that have been observed in some manner inspired by the Bellman equation, with the two algorithms being differentiated by how precisely they do this, and then deriving a (hopefully) improved policy from this modified function. A complete description of Q-learning for a finite MDP is given in Algorithm 5.1.

Initially proposed by Watkins in his PhD thesis in 1989 [100], the discovery of Q-learning was one of the seminal events in modern reinforcement learning. Watkins revisited the algorithm with Dayan a few years later and they were able to establish that the algorithm converges to the optimal action-value function so long as all state-action pairs are visited infinitely many times in the limit and certain reasonable assumptions hold for the step sizes α [99]. An alternative and more general convergence proof was given by Tsitsiklis [90].

Suppose we encounter a state s for which there are many actions a such that $Q(s,a) = 0$ but our estimated action-value functions for those state-action pairs are uncertain, with some being above zero and some below. So although the maximum of the true values is zero, the maximum of our estimated values is

Algorithm 5.1: Q-learning.

```
/* Inputs and outputs                                                    */
```
Input : Discount factor γ
Constant step size parameter α
Action-value function $Q(s, a)$, arbitrary for all states
$s \neq$ terminal state and actions $a \in \mathcal{A}(s)$, with
$Q(\text{terminal state}, \cdot) = 0$

Output: Updated $Q \approx Q^*$

```
/* Start of algorithm proper                                             */
```
foreach *episode* **do**
  Observe current state s
  Choose action a using some policy derived from Q (e.g., ϵ-greedy)
  foreach *step* **of** *episode* **do**
    Take action a and observe cost c and next state s'
    $Q(s, a) = Q(s, a) + \alpha\big[c + \gamma \min_{a'} Q(s', a') - Q(s, a)\big]$
    $s = s'$; $a = a'$
  end
end

positive; this introduces a positive bias, and therefore this problem is known as *maximization bias*.

We can see immediately from Algorithm 5.1 that Q-learning requires a maximization and so may be susceptible to maximization bias. One way to avoid maximization bias is through *double learning*. The basic idea is that we divide our sampled experiences into two sets and use each to find separate estimates for the value function for all actions, which we can call $\hat{Q}_1$ and $\hat{Q}_2$. The trick then is to use one estimate, say $\hat{Q}_1$, to determine the optimal action $a^* = \arg\max_a \hat{Q}_1(a)$ and the other to estimate its actual value through $\hat{Q}_2(a^*) = \hat{Q}_2(\arg\max_a \hat{Q}_1(a))$. Because $\mathbb{E}[\hat{Q}_2(a^*)] = Q(a^*)$, where Q is the true value function, this estimate is unbiased. Similarly, we can repeat the process with the roles of $\hat{Q}_1$ and $\hat{Q}_2$ reversed to obtain another unbiased estimate.

One thing we can note is that using this procedure gives us two estimates but only one is actually updated at each time step, which means that although double learning requires twice the memory as before, the amount of computation at each step stays the same. A double Q-learning method is described by Algorithm 5.2, which is a slightly modified version of that given in [81, p. 144].

We need experience in order to learn optimal value functions and policies but in order to actually gain this experience we must follow some policy. Here we can make a fundamental distinction between at least two classes of reinforcement learning algorithms. *On-policy* methods only update the policy which is actually being followed, whereas *off-policy* methods use *two* policies, one that is updated

Algorithm 5.2: Double Q-learning.

```
/* Inputs and outputs                                                    */
```
Input : Discount factor γ
Constant step size parameter α
Action-value functions $Q_1(s,a)$ and $Q_2(s,a)$, arbitrary for all states $s \neq$ terminal state and actions $a \in \mathcal{A}(s)$, with $Q_1(\text{terminal state},\cdot) = 0$ and $Q_2(\text{terminal state},\cdot) = 0$.
Output: Updated $Q_1 \approx Q^*$ and $Q_2 \approx Q^*$
```
/* Start of algorithm proper                                             */
```
foreach *episode* **do**
  Observe current state s
  Choose action a using some policy derived from Q_1 and Q_2 (e.g., ϵ-greedy in $Q_1 + Q_2$)
  foreach *step of episode* **do**
    Take action a and observe cost c and next state s'
    $r =$ random number from $(0, 1)$
    if $r < 0.5$ **then**
      $Q_1(s,a) = Q_1(s,a) + \alpha\big[c + \gamma Q_2\big(s', \arg\min_{a'} Q_1(s',a')\big) - Q_1(s,a)\big]$
    else
      $Q_2(s,a) = Q_2(s,a) + \alpha\big[c + \gamma Q_1\big(s', \arg\min_{a'} Q_2(s',a')\big) - Q_2(s,a)\big]$
    end
    $s = s'$
  end
end

toward optimality (the *target policy*) and another that is used for exploration in order to gather data (the *behavior policy*). Q-learning is an example of an off-policy method. Note that on-policy methods can be considered a kind of off-policy learning in which the target and behavior policies are the same so off-policy methods are in some sense more general.

On-policy methods are the more straightforward of the two because off-policy methods have to account for the fact that the data used to update the target policy comes from a different policy altogether. Indeed, it is not immediately obvious that this can be done at all but in fact this has a fairly robust theoretical basis [81, Chapter. 5]. Off-policy methods tend to have higher variance than on-policy methods for this reason but because of their greater generality are potentially much the more powerful of the two. (This touches on one of the most important issues in all of machine learning, the *bias-variance tradeoff* [30].)

5.1.4 Balancing exploration and exploitation

Another important issue underlying all of reinforcement learning is what is called the *exploration versus exploitation* problem. Simply put, how do we explore as many actions as possible in order to ensure we find the truly optimal one while also exploiting the best ones we have seen so far?

A generic strategy for solving this problem is what are called ϵ-*greedy* methods. The basic idea is that for some small ϵ, we generate a random number and if it is less than ϵ, we take a random action, and if is not then we choose the best action seen so far (i.e., act greedily). Assuming that we can eventually reach all states from any given starting state, this ensures that given enough time the entire state space will be searched. Typically, we also decrease ϵ over time on the basis that we need less exploration as we gain more of an understanding of the environment and our cost function estimates improve. More mathematically sophisticated techniques for balancing exploration and exploitation such as *upper confidence bounds* (UCB) [40] and *Thompson sampling* [65] have also been proposed, and generally found to outperform ϵ-greedy methods [81, p. 44]. In future work we intend to investigate these alternatives, among others, in the context of task scheduling (see Chapter 6), although so far we have we only considered simple ϵ-greedy methods.

5.1.5 Function approximation

It is clear that for the task scheduling problem in HPC the number of possible states of the environment is likely to be very large. Of course, this depends entirely on exactly how the problem is characterized as a Markov decision process—how we choose to define the states, actions and rewards—but even an informal consideration of the key features which realistically need to be included suggest that this will be the case. Similarly, the number of actions available to us in any

given state obviously depends on how we choose to define them but may well also be large. For example, if the actions available at any time are scheduling a specific task on a specific processor, then the number of available actions at any state is potentially (depending on the precedence constraints) on the order of the product of the number of tasks and the number of processors. Given that modern HPC systems may have millions of processors and the DAGs we are interested in similarly many tasks, it may be impractical to store all possible state-action values $Q(s, a)$ for even a single given state.

One solution to this problem is to find a suitable parameterized function $\tilde{Q}(s, a, w)$ to approximate $Q^*(s, a)$, for all states s and actions $a \in \mathcal{A}(s)$, where w is a finite vector of tunable parameters. The most popular way to do this is using an *artificial neural network*.

5.1.6 Neural networks

Neural networks are widely used for approximating nonlinear functions [81, p. 216]. Borrowing terminology from the biological brains which inspired them, they are networks of connected artificial *neurons*: mathematical functions that receive inputs and apply a nonlinear *activation function* to their weighted sum in order to produce some output, or *activation*. Popular choices for these activation functions are the logistic function,

$$f(x) = \frac{1}{1 + e^{-x}},$$

or the hyperbolic tangent function

$$f(x) = \tanh(x),$$

or the rectifier function

$$f(x) = \max(0, x).$$

Neurons which make use of the latter are called *rectified linear units* or *ReLUs*.

Neural networks are usually structured in *layers*, with an initial input layer of neurons and a final output layer. Any neural network with other *hidden* layers between the two is called a *deep neural network*. In a *feedforward* neural network, external values are used as the inputs for the initial layer of neurons, and the output of these is then used as the inputs for the next layer, and so on; essentially, there are no cycles. On the other hand, if a neural network has at least one cycle, then it is called *recurrent*. A prominent kind of feedforward neural networks are *convolutional neural networks*, which were inspired by how the human brain processes images and therefore tend to excel at tasks such as image recognition [48].

Another important concept here is the *error* or *loss* function. This is a measure of the error in the output of the network (usually, the mean squared error, or log likelihood, or *cross-entropy* [62, Chapter 3]). Typically we want to find the optimal loss function, which minimizes this error. The most popular way to do this for deep neural networks is using an algorithm called *backpropagation*, which calculates the gradient of the loss function with respect to the weights of the network. It is so named because it works backwards, calculating the gradient for the final layer first and then using this result to compute the preceding one, and propagating this information in turn through the whole network. Backpropagation makes heavy use of *gradient descent* and, in particular, *stochastic gradient descent* [49].

Deep neural networks are often referred to as "universal approximators" [81, p. 224] because it has been proven that they are capable of approximating (almost) any function—even with just one hidden layer—provided they satisfy some reasonable conditions [25]. The use of *deep* neural networks for function approximation in reinforcement learning is known as *deep reinforcement learning*; this is the area where most of the notable successes of recent years have been achieved.

5.1.7 Deep reinforcement learning

The idea is to use an approximate value (or action-value) function in place of the true one. But the obvious question is, does this work? Can we still use Q-learning, say, to find an optimal policy if our Q function is an approximation? The answer, in general, is a qualified *yes*, with modifications often needing to be made to the algorithms themselves. For example, in the case of Q-learning, simply plugging an approximate Q-function into Algorithm 5.1 results in the algorithm no longer converging to the optimal action-value function. There are two reasons for this. Firstly, samples of experience are correlated, which is always problematic when fitting data to neural networks as they may implicitly learn to rely on such correlations. Secondly, the number of state-action pairs may be so large that we cannot update the Q function estimate for each of them as was described in Algorithm 5.1. The natural alternative then is to use the *semi-gradient* form of the Q-learning algorithm in which we essentially treat $c+\gamma \max_{a'} Q(s', a', w)$ as a target (called the *TD-target*) and update the action-value function by minimizing the mean-squared error loss

$$\big(c + \gamma \min_{a'} Q(s', a', w) - Q(s, a, w)\big)^2,$$

through stochastic gradient descent (or some similar alternative such as Adam [43]). But the target in this case is non-stationary as it also depends on the network weights w so this will not converge! Fortunately, there are modifications which can be made to the algorithm which can compensate for both of these issues.

One way to handle the problem of correlated samples is *experience replay*, originally proposed by Lin [52]. The basic idea is that we store an agent's previous experiences in a replay memory and then use a random selection from this memory to update the weights of the network. To avoid attempting to learn non-stationary targets, a simple solution is to introduce another network that essentially acts as a target. After a certain number (say, N) of updates have been made to the approximation network, the target network is set with these weights and fixed for the next N episodes of training, with the outputs of the target network used as the Q-learning targets for the updates made in these episodes. Both of these modifications were successfully used by researchers at DeepMind to help train a so-called *Deep Q-Network* (*DQN*) capable of achieving superhuman levels of play on a range of Atari games [57].

Perhaps the most mathematically rigorous treatment of reinforcement learning with approximation is from Bertsekas and Tsitsiklis [9], where most of the main convergence results can be found. However it is safe to say that this is an area in which the theory tends to lag far behind practice with much still to be proven (arguably this is true for all of machine learning). On the other hand, using approximate value functions for reinforcement learning is well-established empirically and has been successfully applied to many real-world problems.

5.2 Similar work

Applying reinforcement learning to scheduling problems—and job shop scheduling in particular—is not a new idea and was proposed at least as far back as Zhang and Dietterich [107]. However interest has increased in the last few years, in large part due to prominent recent successes that have demonstrated the power of using deep neural networks for function approximation, such as the DQN Atari-playing agent already mentioned and the groundbreaking AlphaGo [74] and AlphaGo Zero [75], all from Google DeepMind. In this section we briefly summarize some of the most relevant existing literature concerning the use of RL and/or deep learning for resource management in computing.

Probably the most relevant existing research is that of Wu, Wu, Zhuang and Cheng [104] who train a deep neural network to schedule the tasks of a DAG using the classic REINFORCE algorithm [102]. This work has more recently been extended with Liu [21] to also incorporate *Monte Carlo Tree Search* (MCTS) methods [15]. In both cases the authors compare the quality of the makespan obtained by their scheduler to HEFT and CPOP for randomly generated DAGs, finding that it generally outperforms both. They restrict their investigation to the static problem and assume that all tasks have already been given priorities (i.e., the order in which the tasks are to be scheduled has already been determined). Our investigation covers some of the same ground but differs in two key aspects. First, we use consider the use of other RL algorithms (such as Q-learning) rather

than REINFORCE and MCTS. Secondly, although we initially restrict ourselves to the simpler problem of optimally scheduling tasks according to given priorities, we intend to build on this to handle the full static task scheduling problem (i.e., with no priority list given).

Mao, Alizadeh, Menache and Kandula design and evaluate *DeepRM*, a deep reinforcement learning agent intended to handle general dynamic resource management problems in computer systems and networks [54]. Using a variant of the REINFORCE algorithm, they train a deep neural network capable of learning to balance multiple different objectives (such as minimizing total job execution time or reducing processor idle times). Simulations suggest that their agent performs well compared to existing state-of-the-art approaches. A notable limitation of their work is that they do not consider jobs with intra-task dependencies (such as those which can be represented by task DAGs). Modifications to the DeepRM agent and its extension to static problems were later investigated by Ye et al. [106].

Orhean, Pop and Raicu propose an application that they call *Machine Learning Box* (MBox) which offers reinforcement learning based agents for scheduling on distributed systems. Results of some numerical experiments for simplified models (e.g., where all tasks are homogeneous) are provided but the authors acknowledge that the limitations of their approach prevented any further experimentation and it remains impractical for more complex systems [64].

The use of Q-learning for dynamic load balancing in distributed heterogeneous systems was investigated by Parent et al. [67], and later Samreen and Kumar [71], and Tong, Xiao, Li and Li [86]. Li et al. [50] used artificial neural networks to estimate task execution times in their extension of the classic (static and restricted to homogeneous systems) *Modified Critical Path* (MCP) scheduling heuristic [103]. Training neural networks to predict thread performance on the diverse processing cores of a heterogeneous system in order to maximize total throughput is also considered in the more recent work of Nemirovsky et al. [60]. Interestingly, they found that using deep neural networks sometimes led to diminishing returns compared to using more lightweight networks.

Ipek, Mutlu, Martínez and Caruana propose a reinforcement learning based memory controller for multicore processors [36]. Particular attention is given to how their problem was modeled as an MDP and the design of the reward function, which we found interesting and instructive. The controller uses the Sarsa algorithm to learn the action-value function for the problem. Although simulations suggested that it was capable of outperforming the existing approaches at that time, the RL-based controller was never actually implemented on an physical chip because of the prohibitive cost of constructing such hardware.

Training a neural network to predict memory requirements on HPC systems was investigated by Rodrigues, Cunha, Netto and Spriggs [69]. A Q-learning agent is proposed for routing jobs in ad hoc networks in [20]. Tesauro investigates the use of reinforcement leaning agents for managing computer resource allocation

in data centers in [83], as do Vengerov and Iakovlev in [95]. Negi and Kumar considered the use of machine learning to optimize the CPU time allocated to programs in Linux systems [59]. Galstyan, Czajkowski, and Lerman studied the application of reinforcement learning algorithms for resource allocation in grid computing environments [28]. More generally, machine learning approaches have also been investigated for compiler optimization [79, 89] and managing energy usage in data centers [7].

5.3 Why reinforcement learning?

A word of caution is in order at this point. Despite the recent achievements, it is safe to say that real-world applications of deep reinforcement learning remain rare and it is still very much an emerging subject. One of the main reasons for this is that it remains data-intensive; it is no coincidence that reinforcement learning has achieved perhaps its greatest successes with games, where training data is cheap and easy to acquire. The Google DeepMind Atari controller, for example, required the equivalent of around 38 days of training [57]. This is a particularly important consideration in HPC, where the overriding concern is always to minimize computational time in order to reduce operating costs (which can be considerable). The standard approach in RL to gathering training data when real data is difficult or expensive to acquire is simulation. The use of simulation in reinforcement learning is well-established and has a solid theoretical foundation [9], although of course there is no substitute for the real thing when possible.

In Chapter 6 we consider how reinforcement learning can be used to find good quality schedules for a given DAG, on a given target platform. But no matter how far we minimize the amount of training data we need and how much we maximize the utility of what we have, this approach is always likely to be far more expensive than, say, HEFT. While there may well be applications for which the makespan reduction is worth the extra time needed to actually compute a schedule (or applications that need to be executed so often that minor reductions accumulate significantly), there's nothing new here: it has already been shown that other optimization techniques such as genetic algorithms can be similarly expensive but superior to cheaper heuristics. So what does reinforcement learning offer that they do not?

Our answer to this question is that, unlike those other methods, reinforcement learning offers us the potential for *transfer of learning* across many different DAGs and computing environments. The (perhaps impossible) goal here is a RL-based scheduler that, after a short period of training on a set of test applications on a given machine, is able to quickly obtain good schedules for *any* application run on that machine in the future. The scheduler could accomplish this by learning how to best schedule small subgraphs and combine these solutions to get a good quality

schedule for the original task DAG, or by learning "rules of thumb" that provide good guides for previously-unseen DAGs. Certainly, this may be a pipe dream; transferring learning from one task to even a very similar one remains tricky. But advances in recent years suggest that this may now be possible, especially with deep neural networks. Thus one of the major aims of this PhD research going forward will be to investigate if, how and where we can extend knowledge gained from interaction with specific DAGs and platforms to previously-unseen ones.

Chapter 6

RL-based static task scheduling

In this chapter, we describe our ongoing efforts to apply reinforcement learning methods to the (static) task scheduling problem on accelerated computing platforms. This is a work in progress so this account is intended to be more informal than preceding chapters, chronicling the work we have done so far and the issues we have encountered, as well as speculation about potential future directions. Section 6.3 in particular is entirely speculative, in which we discuss an alternative framework that we suspect may in some instances be a more fruitful way to apply reinforcement learning methods to the task scheduling problem than the approaches we have considered so far.

6.1 With a given priority list

To gauge whether reinforcement learning is a viable option for finding good quality (static) schedules on CPU-GPU platforms, we initially focus on the simpler problem of finding a good schedule for a given list of tasks which have already been assigned priorities according to some rule or heuristic. From the numerical experiments detailed in Chapter 4, it is clear that many listing heuristics can struggle because of unforeseen communication penalties due to earlier processor selections, especially when communication costs are high. So a method for finding good schedules given that task priorities have already been determined would still be useful. As noted in Section 5.2, very similar work along these lines has been done before in [104] and [21] so our intention here is twofold: first, to consider the use of alternative RL algorithms and methods, such as Q-learning; and second, to use this as a base with an eye to possible future extensions, initially to static scheduling agents which operate directly on the DAG itself (i.e., without given priorities) and then to the stochastic DAG scheduling problem.

6.1.1 Characterizing the MDP

Implicit in the assumption that we can apply reinforcement learning to a problem is that we can characterize it as a Markov decision process, with states, actions and costs defined in a useful way that allow us to actually solve the problem that we want to solve. The importance of this can sometimes be understated. Looking at the field of reinforcement learning as a whole, one often gets the impression that finding good choices for these is more of an art than a science. Of course, sometimes there are obvious choices that turn out to also work well, but often it may be unclear, for example, as to why one way to describe the state of an environment works better than another that appears to be just as accurate a summary of the relevant aspects of the problem. Certainly we have found that even seemingly minor changes to how states or costs were defined could have a massive effect on overall performance.

States

Roughly speaking, in any planning problem the state of the system at any time should tell us all the information we need in order to make a decision. We are trying to find the best way to schedule a given list of tasks with priorities. It seems logical then to discretize the problem in such a way that each step corresponds to the scheduling of a single task t (i.e., whichever of the unscheduled tasks has the highest priority) and the decision we want to make is which processor to schedule it on.

In [104], Wu, Wu, Zhuang and Cheng represented the state of the system at step t (i.e., when t is the task to be scheduled) by

$$s_t = \big(n, EST(t, p_1), \ldots, EST(t, p_q), w(t, p_1), \ldots, w(t, p_q)\big), \tag{6.1}$$

where n is the number of tasks that remain unscheduled, $EST(t, p_i)$ is the earliest start time for task t on processor p_i, and $w(t, p_i)$ is its execution cost on that same processor. This proved successful in their simulations and is fairly compact.

In our own testing we have typically defined our state space in a very similar way, although we have experimented widely here. In general, we have been able to achieve the same level of performance as (6.1) with a more compact state space such as

$$s_t = \big(n, EFT(t, p_1), \ldots, EFT(t, p_q)\big),$$

or even in some cases just $s_t = n$. However, given some of the difficulties we have encountered (see below), we continue to search for alternatives that may lead to superior overall performance (or the same level of performance with a more compact state space).

Actions

Given the way we have discretized the MDP, the most reasonable way to define an action is scheduling the task currently under consideration on a specific processor, and this is what we have typically done so far. The only alternative we have considered in any depth is a binary choice between CPU and GPU, with another level of selection (possibly based on an independent cost function) then taking place among all processing resources of that type; we have not pursued this very far as of yet but intend to do so in the future.

Costs

Choosing a useful cost (or reward) function that encourages the agent to do precisely what we want it do can be notoriously tricky [37, 68]. Considering the problem from a dynamic programming perspective however, the sum of all immediate costs should be the total cost incurred over all time—i.e., the makespan. So the natural way to define the cost of an action in a given state is how much that action increments the current makespan, and in our testing so far we have almost always done so. (This is also how costs were defined in [21] and [104].)

6.1.2 Q-learning

In this section, we describe the progress we have made thus far in using Q-learning to learn a good schedule for a given prioritized list of tasks. We chose Q-learning in particular because of its good convergence properties and the impressive results achieved in recent years when neural networks are used to approximate the Q-function, such as in the DeepMind Atari controller [57].

We first began with some small-scale numerical experiments using traditional tabular Q-learning, as described in Algorithm 5.1. Memory constraints restricted these to small DAGs and target platforms with only a few processing resources. These were exploratory tests so we won't present any results here but our experience overall was that the major issue with the scheduling agent was getting it to reliably converge to a good schedule—often the makespan would continue to oscillate, albeit in a relatively narrow band, no matter how many episodes of training were performed. In terms of the actual value of the makespan, results were somewhat mixed: the RL-based scheduling agent almost always at least matched the minimal serial time (which was close to optimal for the small DAGs considered) but was also usually worse than the corresponding HEFT makespan. Still, we consider this fairly promising and it suggests that our characterization of the MDP is along broadly the right lines at least.

To deal with larger problems we need to approximate the cost function. Following the lead of the DeepMind teams, and others, we chose to use a deep neural network. As noted in Section 5.1.5, there are potential issues with this

due to correlations within episode data and the non-stationary nature of the error minimization but remedies for these have successfully been applied elsewhere. In particular, we introduce a *target network* to alleviate the first issue and use *experience replay* to compensate for the latter. A high-level description of our modified Q-learning algorithm for training the network (i.e., approximating the optimal Q function) is given by Algorithm 6.1.

Algorithm 6.1: Deep Q-learning.

```
/* Inputs and outputs                                              */
```
Input : Discount factor γ
Exploration factor ϵ
Approximation network Q and target network Q_t
Experience replay memory buffer R
Number of overall iterations N_I
Number of iterations with fixed Q-target N_Q
Number of episodes of experience per iteration E_Q

Output: Updated $Q \approx Q^*$

```
/* Start of algorithm proper                                       */
```
foreach $i = 1, \ldots, N_I$ **do**
  $Q_t = Q$
  foreach $j = 1, \ldots, N_Q$ **do**
    Run E_Q episodes with an ϵ-greedy policy derived from Q
    Add episode data to replay memory
    Compute TD-targets y_j using Q_t, γ and replay memory
    Update weights of Q by fitting to TD-targets
  end
end

Within this framework, there is a lot of scope and we make the following comments about the choices we have made in our testing so far.

- We use a simple feedforward neural network with only one hidden layer, implemented in the `Keras` package. Experiments with deeper networks have not proven to be advantageous. All neurons in the first two layers use tanh activation functions and neurons in the output layer *softplus* [27]. Adam is used as the optimizer for data fitting. Each unit in the input layer corresponds to a component of the state space and each unit in the output layer to one of the processing units (q altogether), so we typically set the number of neurons in the hidden layer to be $3q$, although many of the other choices we have tried lead to similar performance.

- Episode data is stored as a collection of (state, action, cost, next state) tuples. We typically run all episodes until completion (i.e., when all tasks

in the priority list have been scheduled) as the DAGs we have considered so far have been relatively small.

- With regards to the replay memory, we have experimented with different possibilities, such as its size. In particular, purging older data periodically seems to occasionally be useful, although this isn't entirely clear.

- So far we have only used simple ϵ-greedy methods for exploration, with a few tweaks such as gradually decreasing ϵ to reduce exploration as the network improves. In future we intend to consider UCB methods and Thompson sampling as alternatives.

- TD-targets y_t are computed for each (state, action, cost, next state) = (s, a, c, s') data point in the replay memory by setting $y_t = Q_t(s, \cdot)$, where $Q_t(s, \cdot)$ is the estimate computed by the target network, and modifying the component corresponding to the action a to be

$$y_j^a = c + \gamma \min_{a'} Q_t(s', a'). \tag{6.2}$$

- One notable modification to the algorithm as described that we have considered is initially fitting the network to data corresponding to the HEFT schedule. This seems to lead to better performance (compared to not doing so) but due to oscillations the final makespan can sometimes be worse than the initial HEFT makespan. This is obviously unacceptable and suggests that there could be an underlying issue with our approach that we have thus far overlooked.

- As noted in Section 5.1.3, double Q-learning is often used to avoid maximization bias. This can be extended from the simple tabular version given in Algorithm 5.2 by, for example, replacing (6.2) with

$$y_j^a = c + \gamma Q_t(s', \arg\min_a Q(s', a)),$$

where Q is the most recent version of the approximation network available [92]. Double learning of this form has often been found to lead to superior performance so this is an avenue we intend to pursue in the future.

As for the tabular algorithm, results from our numerical testing so far have been mixed and thus we are actively working on improving performance. The DAGs and target platforms we have considered are relatively small (with ≤ 50 tasks and ≤ 5 processing resources) so the minimal serial time is usually quite close to the schedule makespan computed by listing heuristics, such as HEFT, and (presumably) the optimal makespan. The Q-learning agent typically converges to a schedule fairly quickly (usually before we have run 100 episodes) but this

tends to be either the minimal serial policy or another schedule with a makespan that falls between the minimal serial time and, for example, the HEFT schedule makespan, although we have also seen examples for which the Q-learning schedule is superior to both. Again, this is a work in progress so we are hopeful that we will eventually be able to overcome this issue by making modifications to the algorithm.

6.1.3 REINFORCE

Before we became aware of the work of Wu, Wu, Zhuang and Cheng [104], we had also considered using REINFORCE, and we continued afterward in an attempt to duplicate their findings. However, our results so far have been much less promising and we have encountered many of the same issues as described for Q-learning. It is possible that this may ultimately be down to differences between our computing platform model and that used by Wu, Wu, Zhuang and Cheng. Investigations along these lines are ongoing.

6.2 Without a priority list

The immediate issue with scheduling tasks sequentially according to a given priority list is that it may restrict the quality of the schedule by preventing the optimal scheduling of tasks with low priorities—essentially, the schedule may only be as good as the priority list itself. What we really want is a scheduling agent that takes the actual DAG as input and determines a good schedule from it alone. In this section we discuss one of the RL-based approaches we have considered for this problem. This research is at an early stage so the method may ultimately prove not to be fruitful. Hence this section is intended more to be a sketch of a path we plan to follow rather than an account of the work we have actually done already.

6.2.1 TD-EFT

Recall from Section 4.1 that we can compute a critical path of a DAG using the *optimistic finish time* (OFT) of all tasks across all processing resources, the best possible time that they can possibly be completed, disregarding all processor contention. More specifically, for all tasks and processors we compute

$$OFT(t,p) = w(t,p), \quad \text{if } t \text{ is an entry task,}$$
$$OFT(t,p) = w(t,p) + \max_{s \in P(t)} \left\{ \min_{p'} \{ w(s,p') + c_{st} \} \right\}, \quad \text{otherwise.}$$

In reality of course the *actual finish time* (AFT) when scheduling task s on a processor a, $AFT(s,a)$ depends on the earliest time $R(p,t)$ that the processor p

can actually execute task t, i.e.,

$$AFT(t,p) = w(t,p) + R(p,t), \quad \text{if } s \text{ is an entry task,}$$
$$AFT(t,p) = w(t,p) + \max\left(R(p,t), \max_{s \in P(t)}\left\{\min_{p'}\{w(s,p') + c_{st}\}\right\}\right), \quad \text{otherwise.}$$

The idea behind our *Temporal Difference—Earliest Finish Time* (TD-EFT) scheduling algorithm is to use this as the basis for a temporal difference learning scheme. In particular, we initialize a state-action function Q such that

$$Q(t,p) = OFT(t,p) \quad \forall t, p.$$

Then we derive a priority list of the tasks from this Q-function using the same method described in Section 4.2.2 to compute the *optimistic* weighting in the HEFT function—i.e., set the weights in an approximation DAG according to where the Q-function suggests it should be scheduled, calculate communication costs based on this and then compute the upward rank of all tasks.

Once the priority list has been computed, we run an episode (i.e., schedule the DAG) in this order, with all tasks t scheduled on the processor p_t such that

$$p_t := \arg\min_p Q(t,p),$$

at the earliest possible time that p_t can actually execute it. For each task we store the tuple (p_t, start time, finish time), which we call the *trace*. We then use this to update the Q function according to a temporal difference update of the form

$$Q(t,p) \mathrel{+}= \alpha\left[C(t,p) + \gamma \max_{s \in P(t)} \min_{p'}\{Q(s,p') + c_{ts}\} - Q(t,p)\right],$$

where $C(t,p)$ is a cost function that encodes how "good" the actual start time of t was relative to the optimistic earliest time it could have started (according to Q). The obvious question here then is how we define C. We have experimented with several different choices. The best-performing so far is to define

$$C(t,p) = 0, \quad \text{if } t \text{ is an entry task,}$$
$$C(t,p) = AST(t,p) - \max_{s \in P(t)} AFT(s), \quad \text{otherwise.}$$

After the Q-function has been updated we run another episode, then use the data from that to update the function again, and so on for a specified number of iterations, or until the makespan of the derived policy converges.

Results so far with this method are similar to the Q-learning priority list scheduler discussed in 6.1.2: typically the schedule makespan derived is "in the

right ballpark" but worse than, say, the HEFT makespan. In fact results so far are a little worse than for the Q-learning scheduler in that, while the makespan often falls between the HEFT makespan and the minimal serial time, it is also often actually worse than the latter (i.e., we would regard it as a failure). However, as noted at the beginning of this chapter, this is a work in progress so we are still actively experimenting and are cautiously optimistic that this approach, or something very similar, will yield dividends in the future.

Another issue is that the algorithm is currently restricted to small DAGs and target platforms because of the initialization of the Q-function with the optimistic finish times, which can be very expensive to compute. Again, we intend to investigate possible alternative approaches which are cheaper to compute in the future.

6.3 RL-augmented scheduling

The difficulties we have encountered so far in successfully applying reinforcement learning methods directly to the task scheduling problem suggest that a change in our focus may be in order. Although we remain optimistic that we can improve the performance of the methods described in this chapter so far, and there are other ideas that we intend to investigate in future, we have come to believe that the most practical use for reinforcement learning in the context of task scheduling may be for augmenting and guiding existing methods, suggest as HEFT, rather than as an end-to-end scheduling agent. This would potentially ameliorate one of the major drawbacks of reinforcement learning, namely the cost, since (hopefully) less data will be required in this case. In addition, under this paradigm the goal would change to learning general rules and guidelines rather than optimal policies, so there is much more potential for transferring this learning across different DAGs and computing platforms.

For example, the numerical experiments described in Chapter 4.2 suggested to us that in some circumstances it may be more useful in the long run, rather than learning the optimal way to act, to learn which actions we definitely *shouldn't* take, such as when communication costs are very high and the best way to ensure good performance is to avoid incurring them as far as possible. Similarly, in uncertain environments—such as the stochastic DAG scheduling problem—the most effective policy in the long run may be to act conservatively and avoid taking actions that are usually good but can with some non-negligible probability be be detrimental, such as scheduling a task on a processor with execution times that have a low mean but very high variance.

Chapter 7

Future work

Although we have made occasional comments in previous chapters, here we elaborate more fully on our plans for the remainder of this PhD project. These may change depending on, for example, interesting results which suggest we should divert our focus elsewhere but this is the road map that we currently intend to follow.

7.1 Recap of progress so far

Before discussing the work we intend to do over the next two years, we briefly summarize what we have done already. We have:

- Implemented a simple but flexible software framework for simulating the scheduling of DAG applications on user-defined CPU-GPU platforms. In addition, this framework also allows us to easily implement and evaluate existing scheduling heuristics, as well as prototype novel approaches of our own.
- Evaluated several existing (static) scheduling heuristics for CPU-GPU environments through simulation and considered possible optimizations.
- Investigated how reinforcement learning can be applied to several different aspects of the static DAG scheduling problem.

While our overall progress has certainly not been as rapid as we would like and we would ideally like to have more encouraging results at this point, we do believe that this gives us a fairly good base to build on for future research.

7.2 Extend comparison of listing heuristics

There are many aspects of our investigation detailed in Chapter 4 that we would like to pursue further. For example, we want to further investigate lookahead

heuristics, especially the possibility of using sampling-based methods to improve efficiency. We also want to consider task duplication heuristics to some extent as this is another alternative solution for problematic high-data environments. There are also more minor things that we wish to do to improve the study, such as extending PEFT to larger DAGs, which may require more powerful computational resources than we have used so far.

In addition, aspects of the study should ideally be repeated with different underlying assumptions, such as when task CPU and GPU times are not decided entirely by theoretical peak processor speeds or when tasks do not transfer the same data amounts to all of their children. From the perspective of a possible publication, directly comparing the heuristics for these different cases may be more interesting than the current approach of considering each of the heuristics separately and investigating possible optimizations before making comparisons between them.

7.3 RL-based scheduling

Given that results with the RL-based scheduling agents that we have considered so far have been mixed at best, we first need to modify these algorithms to improve performance, or else establish that they are not a fruitful approach and ideally identify why. Work along these lines continues and we are cautiously optimistic that it will ultimately be successful, given that [21] and [104] have shown that it is possible in at least some cases. However completely new approaches may have to be considered instead if we cannot achieve success here in the near future.

There are also other RL-based methods that we wish to consider, either in an end-to-end scheduling agent or as a means for augmenting existing heuristics. In particular, Monte Carlo Tree Search (MCTS) methods seem to us an avenue worth investigating in several different contexts, such as determining the order in which independent groups of tasks should be scheduled (a subproblem that occurs frequently in dynamic scheduling and, for example, the HBMCT heuristic) or as an alternative processor selection method in listing heuristics.

7.4 Stochastic and dynamic scheduling

As stated in Section 1.4, we have always intended to extend this work to the stochastic scheduling problem, where computation and communication times are not known precisely before execution, as this is much more realistic than the static alternative. There is a lot of overlap here with the previous section as we believe RL methods may be particularly useful, given that learning how to optimally act in uncertain environments is the fundamental goal of reinforcement learning. In particular we intend to investigate the use of RL to modify static

schedules for the stochastic problem, similar to the MCS heuristic of Zheng and Sakellariou [110], as well as extend the methods we have developed for the static problem.

7.5 Transfer of learning

In Section 5.3, the main reason we gave for why we believe reinforcement learning to be an especially promising approach for the task scheduling problem is because of the potential for transferring learning between different DAGs and platforms. Conversely, achieving this may be the only way it will ever be practical. Assuming for the moment that we can successfully improve the algorithms discussed in Chapter 6 so that they consistently obtain schedules superior to, say, HEFT, it is very likely that they will still be significantly more expensive. Of course there may be situations in which the extra time spent obtaining an optimal schedule is worth it in the long run, such as when the same application must be executed many times, but in general this may not be the case. Using RL to augment cheaper heuristics, as discussed in Section 6.3, is likely to be cheaper but may still be prohibitively expensive unless we can incorporate some degree of learning transfer.

There are many things we wish to consider here. In particular, we plan to investigate possible heuristic approaches for combining (known) optimal schedules for constituent subsets of the nodes of a DAG in order to construct good schedules for the entire DAG. This was also entail studying the related problem of identifying substructures of the DAG for which good schedules are known, which is in general NP-hard itself. On a simpler level, we also want to investigate if/how an RL-based scheduling agent can apply learning from one application DAG to another which represents that same application but is, for example, of a different size (e.g., Cholesky DAGs representing different tile granularities of a matrix).

7.6 Provisional timetable for next year

The following is a timetable which gives a rough outline of what we currently believe will be the structure of the next year of this research.

Present–December 2019

- Complete work identified in Section 7.2, with the aim that this will form the bulk of a possible publication.
- Conclude investigation/optimization of deep Q-learning scheduling agent described in Section 6.1.2. If unable to achieve a satisfactory scheduling agent with this approach, explain why this the case.

- Similarly, improve performance of TD-EFT algorithm (see Section 6.2.1) or adequately explain why it does not work.
- Build on experience gained with the previous two points to implement and evaluate other RL-based scheduling algorithms, including perhaps both some which operate directly on an input DAG and some which learn how to schedule a given priority list of tasks.
- Continue to investigate how RL can be used to optimize existing scheduling heuristics.
- Extend simulator software to allow us to consider stochastic DAG scheduling.
- Investigate how static schedules can be extended to stochastic environments, using RL-based methods and otherwise.
- Likewise, investigate how static scheduling heuristics and RL-based agents can be extended for stochastic DAG scheduling.

January–June 2020

- Continue development of RL-based stochastic scheduling agents.
- Consider how we can transfer learning to enable more practical RL-based scheduling.

References

[1] 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 *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, IEEE, 2011, pages 932–943.

[2] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, 2009.

[3] H. Arabnejad and J. G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):682–694, 2014.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[5] 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.

[6] Blaise Barney. Message Passing Interface (MPI). [Online; accessed 05-April-2019].

[7] Josep Ll. Berral, Íñigo Goiri, Ramón Nou, Ferran Julià, Jordi Guitart, Ricard Gavaldà, and Jordi Torres. Towards energy-aware scheduling in data centers using machine learning. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, New York, NY, USA, 2010, pages 215–224. ACM.

[8] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, MA, 1995.

[9] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, 1996. 512 pp. ISBN 1-886529-10-8.

[10] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. DAG scheduling using a lookahead variant of the Heterogeneous Earliest Finish Time algorithm. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2010, pages 27–34.

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, May 2011, pages 1432–1441.

[12] 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.

[13] T. D. Braun, H. J. Siegal, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, Bin Yao, D. Hensgen, and R. F. Freund. A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, 1999, pages 15–29.

[14] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810 – 837, 2001.

[15] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[16] L. Canon and E. Jeannot. Evaluation and optimization of the robustness of DAG schedules in heterogeneous environments. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):532–546, 2010.

[17] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative evaluation of the robustness of DAG scheduling heuristics. In *Grid Computing*, Springer, 2008, pages 73–84.

[18] Louis-Claude Canon, Loris Marchal, Bertrand Simon, and Frédéric Vivien. Online scheduling of task graphs on hybrid platforms. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, Cham, 2018, pages 192–204. Springer International Publishing.

[19] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[20] Yu-Han Chang, Tracey Ho, and Leslie Pack Kaelbling. Mobilized ad-hoc networks: A reinforcement learning approach. In *Autonomic Computing, 2004. Proceedings. International Conference on*, IEEE, 2004, pages 240–247.

[21] Yuxia Cheng, Zhiwei Wu, Kui Liu, Qing Wu, and Yu Wang. Smart DAG tasks scheduling between trusted and untrusted entities using the MCTS method. *Sustainability*, 11(7), 2019.

[22] François Chollet et al. Keras: The Python deep learning library. [Online; accessed 06-December-2018].

[23] Kallia Chronaki, Alejandro Rico, Marc Casas, Miquel Moretó, Rosa M. Badia, Eduard Ayguadé, Jesus Labarta, and Mateo Valero. Task scheduling techniques for asymmetric multi-core systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2074–2087, 2017.

[24] Gennaro Cordasco, Rosario De Chiara, and Arnold L. Rosenberg. Assessing the computational benefits of AREA-oriented DAG-scheduling. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, Berlin, Heidelberg, 2011, pages 180–192. Springer Berlin Heidelberg.

[25] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[26] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, London, UK, UK, 1993, pages 146–160. Springer-Verlag.

[27] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. Incorporating second-order functional knowledge for better option pricing. In *Advances in neural information processing systems*, 2001, pages 472–478.

[28] Aram Galstyan, Karl Czajkowski, and Kristina Lerman. Resource allocation in the grid using reinforcement learning. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3*, AAMAS '04, Washington, DC, USA, 2004, pages 1314–1315. IEEE Computer Society.

[29] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0716710455.

[30] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4(1):1–58, 1992.

[31] Green500. November 2018. [Online; accessed 05-April-2019].

[32] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, 2012, pages 1242–1253.

[33] T. Hagras and J. Janeček. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pages 107–.

[34] Jane N. Hagstrom. Computational complexity of PERT problems. *Networks*, 18(2):139–147.

[35] E. Ilavarasan and P. Thambidurai. Low complexity performance effective task scheduling algorithm for heterogeneous computing environments. *Journal of Computer sciences*, 3(2):94–103, 2007.

[36] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, IEEE, 2008, pages 39–50.

[37] Alex Irpan. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[38] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, 1998.

[39] Michael A. Iverson, Füsun Özgüner, and Gregory J. Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *4th Heterogeneous Computing Workshop (HCW '95)*, 1995, pages 93–100.

[40] Thomas Jaksch, Ronald Ortner, and Peter Auer. Near-optimal regret bounds for reinforcement learning. *Journal of Machine Learning Research*, 11(Apr):1563–1600, 2010.

[41] Hironori Kasahara. Standard Task Graph Set. [Online; accessed 27-November-2018].

[42] James E. Kelley Jr. and Morgan R. Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, Eastern joint IRE-AIEE-ACM computer conference*, ACM, 1959, pages 160–173.

[43] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[44] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[45] Oak Ridge National Laboratory. Summit. [Online; accessed 14-May-2019].

[46] Oak Ridge National Laboratory. Titan. [Online; accessed 14-May-2019].

[47] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[48] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[49] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. *Efficient BackProp*, pages 9–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35289-8.

[50] Jiangtian Li, Xiaosong Ma, Karan Singh, Martin Schulz, Bronis R. de Supinski, and Sally A. McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, IEEE, 2009, pages 89–100.

[51] K. Li, X. Tang, B. Veeravalli, and K. Li. Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems. *IEEE Transactions on Computers*, 64(1):191–204, 2015.

[52] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321, 1992.

[53] Grzegorz Malewicz, Ian Foster, Arnold L. Rosenberg, and Michael Wilde. A tool for prioritizing DAGMan jobs and its evaluation. *Journal of Grid Computing*, 5(2):197–212, 2007.

[54] 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, New York, NY, USA, 2016, pages 50–56. ACM.

[55] Microsoft. Project Catapult. [Online; accessed 05-April-2019].

[56] Sparsh Mittal and Jeffrey S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4):69:1–69:35, 2015.

[57] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[58] W. Nasri and W. Nafti. A new DAG scheduling algorithm for heterogeneous platforms. In *2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing*, Dec 2012, pages 114–119.

[59] Atul Negi and P. Kishore Kumar. Applying machine learning techniques to improve Linux process scheduling. In *TENCON 2005 2005 IEEE Region 10*, IEEE, 2005, pages 1–6.

[60] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal. A machine learning approach for performance prediction and scheduling on heterogeneous CPUs. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2017, pages 121–128.

[61] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, New York, NY, USA, 2008, pages 16:1–16:14. ACM.

[62] Michael A. Nielsen. *Neural networks and deep learning*. Determination Press, 2015.

[63] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[64] 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.

[65] Yi Ouyang, Mukul Gagrani, Ashutosh Nayyar, and Rahul Jain. Learning unknown Markov decision processes: A Thompson sampling approach. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, Curran Associates, Inc., 2017, pages 1333–1342.

[66] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[67] Johan Parent, Katja Verbeeck, Jan Lemeire, Ann Nowe, Kris Steenhaut, and Erik Dirkx. Adaptive load balancing of parallel applications with multi-agent reinforcement learning on heterogeneous systems. *Scientific Programming*, 12(2):71–79, 2004.

[68] Ivaylo Popov, Nicolas Heess, Timothy P. Lillicrap, Roland Hafner, Gabriel Barth-Maron, Matej Vecerik, Thomas Lampe, Yuval Tassa, Tom Erez, and Martin A. Riedmiller. Data-efficient deep reinforcement learning for dexterous manipulation. *CoRR*, abs/1704.03073, 2017.

[69] Eduardo R. Rodrigues, Renato L. F. Cunha, Marco A. S. Netto, and Michael Spriggs. Helping HPC users specify job memory requirements via machine learning. In *Proceedings of the Third International Workshop on HPC User Support Tools*, HUST '16, Piscataway, NJ, USA, 2016, pages 6–13. IEEE Press.

[70] R. Sakellariou and H. Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, pages 111–124.

[71] Faiza Samreen and M. Sikandar Hayat Khiyal. Q-learning scheduler and load balancer for heterogeneous systems. *Journal of Applied Sciences*, 7 (11):1504–1510, 2007.

[72] Vinay Saripalli, Guangyu Sun, Asit Mishra, Yuan Xie, Suman Datta, and Vijaykrishnan Narayanan. Exploiting heterogeneity for energy efficiency in chip multiprocessors. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(2):109–119, 2011.

[73] Karan R. Shetti, Suhaib A. Fahmy, and Timo Bretschneider. Optimization of the HEFT algorithm for a CPU-GPU environment. In *PDCAT*, 2013, pages 212–218.

[74] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[75] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

[76] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. *Concurrency and Computation: Practice and Experience*, 27(16):4075–4090.

[77] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Modeling and simulation of a dynamic task-based runtime system for heterogeneous multi-core architectures. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, Cham, 2014, pages 50–62. Springer International Publishing.

[78] StarPU. StarPU handbook. [Online; accessed 27-May-2018].

[79] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. *SIGPLAN Not.*, 38(5):77–90, 2003.

[80] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[81] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd edition, MIT press Cambridge, 2017.

[82] Xiaoyong Tang, Kenli Li, Guiping Liao, Kui Fang, and Fan Wu. A stochastic scheduling algorithm for precedence constrained tasks on grid. *Future Generation Computer Systems*, 27(8):1083 – 1091, 2011.

[83] Gerald Tesauro. Online resource allocation using decompositional reinforcement learning. In *AAAI*, volume 5, 2005, pages 886–891.

[84] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394.

[85] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[86] Zhao Tong, Zheng Xiao, Kenli Li, and Keqin Li. Proactive scheduling in distributed computing–A reinforcement learning approach. *Journal of Parallel and Distributed Computing*, 74(7):2662 – 2672, 2014. Special Issue on Perspectives on Parallel and Distributed Processing.

[87] Top500. November 2018. [Online; accessed 28-November-2018].

[88] 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.

[89] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. *SIGPLAN Not.*, 44(6):177–187, 2009.

[90] John N. Tsitsiklis. Asynchronous stochastic approximation and q-learning. *Machine Learning*, 16(3):185–202, 1994.

[91] L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189 – 201, 1979.

[92] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. *CoRR*, abs/1509.06461, 2015.

[93] Richard M. van Slyke. Monte Carlo methods and the PERT problem. *Operations Research*, 11(5):839–860, 1963.

[94] Maarten van Steen and Andrew S. Tanenbaum. A brief introduction to distributed systems. *Computing*, 98(10):967–1009, 2016.

[95] David Vengerov and Nikolai Iakovlev. A reinforcement learning framework for dynamic resource allocation: First results. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, IEEE, 2005, pages 339–340.

[96] Ashish Venkat and Dean M. Tullsen. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, IEEE, 2014, pages 121–132.

[97] Chris Walker. New Intel core processor combines high-performance CPU with custom discrete graphics from AMD to enable sleeker, thinner devices. [Online; accessed 05-April-2019].

[98] Lingyuan Wang, Saumil Merchant, and Tarek El-Ghazawi. Exploiting hierarchical parallelism using UPC. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, IEEE, 2011, pages 1216–1224.

[99] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[100] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[101] Wikichip. PEZY-SC2. [Online; accessed 05-April-2019].

[102] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[103] Min-You Wu and Daniel D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE transactions on parallel and distributed systems*, 1(3):330–343, 1990.

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

[105] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical DAG scheduling for hybrid distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, IEEE, 2015, pages 156–165.

[106] Yufei Ye, Xiaoqin Ren, Jin Wang, Lingxiao Xu, Wenxia Guo, Wenqiang Huang, and Wenhong Tian. A new approach for resource scheduling with deep reinforcement learning. *CoRR*, abs/1806.08122, 2018.

[107] Wei Zhang and Thomas G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, 1995, pages 1114–1120.

[108] Henan Zhao and Rizos Sakellariou. An experimental investigation into the rank function of the Heterogeneous Earliest Finish Time scheduling algorithm. In *Euro-Par 2003 Parallel Processing*, Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, Berlin, Heidelberg, 2003, pages 189–194. Springer Berlin Heidelberg.

[109] W. Zheng, L. Tang, and R. Sakellariou. A priority-based scheduling heuristic to maximize parallelism of ready tasks for DAG applications. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pages 596–605.

[110] Wei Zheng and Rizos Sakellariou. Stochastic DAG scheduling using a Monte Carlo approach. *Journal of Parallel and Distributed Computing*, 73(12): 1673 – 1689, 2013. Heterogeneity in Parallel and Distributed Computing.

[111] Mawussi Zounon. Novel methods for static and dynamic scheduling. [Online; accessed 08-May-2019].