The Task Scheduling Problem in High-Performance Computing

First Year PhD Continuation Report

Thomas McSweeney[1]

10th October 2018

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

Table of contents

1 Introduction **4**

2 Trends in high-performance computing **6**

2.1 Heterogeneous architectures 7

2.1.1 Accelerators and coprocessors 8

2.1.2 Heterogeneous multicore processors 9

2.2 Hierarchical parallelism 9

2.3 Distributed computing 9

2.4 Task parallelism 11

2.4.1 From tasks to DAGs 12

2.4.2 Relevant software 14

2.5 The task scheduling problem 15

3 Existing task scheduling heuristics **17**

3.1 Simple heuristics 18

3.1.1 Relating to queuing theory 19

3.2 More complex scheduling strategies 21

3.2.1 Clustering schedulers 21

3.2.2 Guided random schedulers 22

3.2.3 Duplication-based schedulers 23

3.2.4 Listing schedulers 23

4 Introduction to reinforcement learning **28**

4.1 The reinforcement learning framework 29

4.2 Markov decision processes 31

4.2.1 The value and action-value functions 32

4.3 Balancing exploration and exploitation 33

4.4 Credit assignment 35

5 Dynamic programming **37**

5.1 The Bellman equation 38

5.2 Policy evaluation and improvement 39

5.3 Policy iteration 41

5.4 Value iteration . . . 43
5.5 The curse of dimensionality . . . 44

6 Monte Carlo learning **45**
6.1 On-policy and off-policy methods . . . 45
6.2 Monte Carlo algorithms . . . 47
6.2.1 Exploring starts . . . 48
6.2.2 Without exploring starts . . . 51
6.2.3 Off-policy Monte Carlo methods . . . 51

7 Temporal-difference learning **55**
7.1 TD prediction . . . 55
7.2 Sarsa . . . 57
7.3 Q-Learning . . . 57
7.3.1 Expected Sarsa . . . 59
7.4 Double learning . . . 60

8 Function approximation and deep reinforcement learning **62**
8.1 Neural networks and deep learning . . . 63
8.2 Deep reinforcement learning . . . 65
8.3 Notable successes . . . 66
8.3.1 Mastering Atari games . . . 67
8.3.2 AlphaGo . . . 68

9 Outline of our approach and progress so far **69**
9.1 What do we really want? . . . 69
9.2 An immediate problem . . . 70
9.3 Our solution: simulation . . . 71
9.3.1 SimGrid . . . 72
9.4 Some current and potential issues . . . 73
9.4.1 Characterizing the MDP . . . 73
9.4.2 Transfer of learning . . . 75
9.4.3 Remaining practical . . . 75
9.4.4 Designing new algorithms . . . 76
9.5 Related work . . . 76
9.6 Summary of progress so far . . . 77
9.7 Proposed timetable for future research . . . 78

Appendices **80**

Appendix A Personal development **80**

Appendix B Numerical linear algebra primer **83**

Appendix C Code **85**
C.1 `Block Cholesky factorization` 85

References **93**

Chapter 1

Introduction

As the most powerful modern computers approach *exascale*—capable of performing at least 10^{18} floating-point operations a second—the prevailing hardware trends that have allowed them to achieve this performance will also necessitate changes in how we design the programs and algorithms we intend to execute on them, in order to fully exploit their awesome potential. This PhD research focuses on one such issue, concisely summarized by the following question: how do we most efficiently schedule the constituent parts of a large computational job in modern computing environments?

This question is becoming increasingly pertinent. We focus in particular on *high-performance computing* (HPC) systems, i.e., supercomputers, the most powerful machines available today. This is where the problem is most acute, not least because modern HPC systems have annual energy costs well into the millions of dollars and thus must manage their available resources as intelligently as possible. But we should emphasize here that our question is not specific to HPC and we expect it to become more widely relevant as the current hardware trends there filter downward to everyday computing environments.

Increasing *heterogeneity* in HPC architectures 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. The move towards *multicore* processing means that modern processors will each generally have multiple processing cores, which allows us to exploit *parallelism* to a greater degree than ever before. In addition, the growing popularity of *distributed* HPC architectures compounds both of the aforementioned trends. But we can only take advantage of these developments if we fully consider their implications when finding a solution to our question.

In Chapter 2, we give a brief overview of the state of modern HPC systems and discuss the relevant architectural trends in greater detail. We then introduce a programming paradigm that has become popular in recent years as an attempt to harness these hardware changes without complicating the job of the programmer unduly. We conclude the chapter by expressing the problem we are considering

within this programming framework.

Although more important than ever before, this problem is by no means a new one. Many different approaches have been proposed (and tried) before; discussion of these forms the bulk of Chapter 3. We briefly analyze some simple but widely-used heuristics from the perspective of queuing theory. Later in the chapter, we consider in depth the popular *Heterogeneous Earliest Finish Time* (HEFT) algorithm.

The next few chapters then introduce the framework that we propose for tackling our scheduling problem: *reinforcement learning*. This is a kind of machine learning that has been successfully applied to many difficult, seemingly intractable problems in recent years.

In Chapter 4 we give a general introduction to reinforcement learning, and frame the subject more formally in terms of learning a *Markov decision process* with unknown dynamics. We also discuss some generic problems we face and some of the simple methods employed to try and solve them.

Reinforcement learning is inextricably linked with *dynamic programming*, a powerful family of algorithms for solving stochastic optimal control problems. In Chapter 5, we briefly run through the basics of dynamic programming, and introduce some fundamental methods that form the basis for many of the reinforcement learning algorithms introduced in later chapters.

In Chapter 6 we then introduce so-called *Monte Carlo* reinforcement learning, where we learn optimal ways to behave based on data gathered from samples of experience. We then describe some simple algorithms that have proven to be both extremely powerful and popular.

Similarly, in Chapter 7, we introduce *temporal-difference* learning, a family of methods that combine some of the features of Monte Carlo learning with those of classical dynamic programming.

Interest in reinforcement learning has exploded in recent years in large part due to notable successes achieved by combining it with another booming area of machine learning, *deep learning*. This is the focus of Chapter 8.

In the remaining chapter, Chapter 9, we then shift the emphasis to summarizing the progress we have made in the first year of this PhD research project, describing also the problems we have encountered and the work we intend to do in the future, including a proposed timetable for the remainder of the project.

Chapter 2

Trends in high-performance computing

About fifteen years ago, it became apparent that although Moore's Law for transistor densities continued to hold true, performance improvement of single-core processors had begun to stagnate [20]. This was primarily because physical upper limits on their clock frequencies had been reached due to heat, power and current leakage issues [5, 127], a problem sometimes referred to as the *frequency wall*. To get around this, designers instead began to increase the number of processing cores on each individual chip.

Thus we entered the multicore[1] era. Today, processors with more than one thousand cores are not unheard of [19], and even mainstream computers—desktops, laptops, tablets, games consoles—now almost universally have multicore processors. Modern HPC systems themselves may have hundreds of thousands or even millions of cores: the Sunway TaihuLight—as of this writing, the world's fastest supercomputer—has 40,690 processors, each with 260 cores, for a total of 10,579,400 cores [137].

Although the move towards multicore processing enabled HPC systems to surmount the frequency wall and maintain constant improvement in performance, energy issues soon came to the forefront. As processors have grown more and more powerful, their energy consumption and heat production has grown correspondingly, to an impractical degree [39]; this problem is often called the *energy wall* [26]. Computing systems with many different kinds of processors have thus became increasingly attractive. The basic idea is that by assigning vital or compute-intensive tasks to higher performance (and therefore usually power) processors and less pressing or compute-intensive tasks to lower performance (power)

[1] A distinction is often 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 that generally have upwards of dozens of cores. However, as they are simply a kind of multicore processor, we will use that term unless we feel the need to distinguish between them.

ones, we can reduce wasted computational effort and therefore improve overall energy efficiency [26, 47]. Modern high-performance computing architectures are increasingly heterogeneous and it is expected that this trend will, if anything, accelerate in the future [7, 20]. We discuss this in greater depth in Section 2.1.

Of course, the massive growth in the number of cores available to us offers us massively more potential for performing parallel computations. But it also means that now almost everything must be parallelizable in order to make the most of the resources available, thus making parallelism more central than ever before. Indeed, we shall see in Section 2.2 that modern HPC systems potentially offer many different levels of parallelism, increasing its importance still further.

Another great sea change in HPC in recent years has been the increasing prominence of *distributed* architectures, in which we may have many separate computational *nodes* working in tandem. These nodes may be very different from one another and separated by great distances, potentially being oceans apart. Distributed computing has been in vogue for a long time but the complexity of modern systems has increased to the extent that it deserves to be discussed in more detail, which we do in Section 2.3.

The massive growth in parallelism afforded to us by modern HPC systems is ultimately problematic for the simple reason that humans find it very difficult to think in parallel and so our software and algorithms lag far behind the hardware. Fundamental changes to our traditional programming frameworks are therefore required. One such approach is *task-based parallel programming*, the focus of Section 2.4.

We conclude this chapter in Section 2.5 by summarizing the trends we have highlighted in the previous sections, as well as restating the problem which motivates this research in the context of the task-based programming framework.

2.1 Heterogeneous architectures

Of course, although energy concerns may have instigated the move towards heterogeneous architectures, they also simply make sense on a much more fundamental level. Different kinds of processors have different attributes, such as their power, energy consumption and, crucially, what sort of computational tasks they are good at. So if we want to achieve the best possible performance, why don't we make sure that we perform each of the constituent parts of an application on the kind of processor best suited to handle it?

Recent years have seen a sharp increase in the prevalence of HPC systems which make use of *accelerators*, dedicated hardware used for increased performance, or *coprocessors*, additional processing units designed to supplement the traditional CPUs. These are the subject of Section 2.1.1. Initially, research was often focused either on how they could be used as a replacement for more traditional processors, or how certain parts of an application could be offloaded onto

them (with relatively little coordination with the rest of the processors). However, the consensus is now shifting toward the idea that to achieve the best possible performance we need to design systems that seamlessly integrate a wide variety of processing units [78], by which we mean any device capable of performing general computations.

It has not escaped notice that the 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. We shall touch on these briefly in Section 2.1.2.

2.1.1 Accelerators and coprocessors

More than one hundred of the machines on the November 2017 TOP500 list of the world's most powerful supercomputers are accelerated[2] systems [137]. 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, we find that all of the current top ten are accelerated systems [46].

There are a wide variety of devices employed as accelerators or coprocessors in modern HPC but the real driver of this trend in the last few years has been the *Graphics Processing Unit* (GPU[3]). Fueled by their importance in the gaming industry, GPUs have become both increasingly cheap and powerful in recent years and, being optimized for graphics rendering applications, they have proven to be adept at many other large, parallel tasks which access memory in a regular manner [91].

For example, in numerical linear algebra, GPUs have often been found to deliver far better performance than CPUs when multiplying or factorizing very large matrices—which is a classic *embarrassingly parallel* task—but the converse is usually true when performing smaller, more serial tasks [3, 156].

Even more so than linear algebra, the area most responsible for the recent popularity of GPUs in HPC is machine learning, a field that has attracted tremendous interest in recent years and will be treated in later chapters of this report. This is largely because in such applications we generally have to do lots of computations (which often can be done in parallel) on an enormous amount of data, which is exactly the kind of task for which GPUs are intrinsically well-suited [123].

Also popular as accelerators are *Field-Programmable Gate Arrays* (FPGAs), integrated chips that can be programmed for specialized tasks. They have been around for several decades but have recently become more prominent than ever before, as evidenced by Microsoft's Project Catapult, where they are used as accelerators in network servers in the company's data centers [76].

[2]By which we also include those making use of coprocessors.

[3] 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.

Many HPC systems simply use powerful manycore processors as accelerators rather than GPUs or FPGAs. Indeed, despite the overall dominance of GPU-based accelerator systems on the most recent Green500 list, four of the top five machines actually use the 2048-core PEZY-SC2 [153] processor as an accelerator [137].

2.1.2 Heterogeneous multicore processors

Clearly the same principle which motivates the trend toward architectures with a variety of different processing units extends further to cores on the same chip. Heterogeneous multicore processors are becoming increasingly common and offer promising performance relative to traditional homogeneous architectures, particularly with regard to energy efficiency [62, 110, 145].

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 this is expected to become the default in the near future [146].

2.2 Hierarchical parallelism

High-performance computers have been almost universally parallel since at least the 1990s, but modern machines take this to extremes never seen before. Increasingly, we have a *hierarchy* of potential *parallelism*: across different nodes in a system, between the processing units within those nodes themselves, among the cores of *those* processors, and so on.

Figure 2.1 illustrates the hierarchical parallelism of IBM's Blue Gene series of supercomputers. We see that we have two cores to a chip, two chips to a compute card, sixteen compute cards to a node code, and so on, for multiple levels of parallelism. This is typical of a modern HPC system.

As previously noted, the most significant problem with parallel computing in general is that it has traditionally proven difficult from a programming perspective [32, 84]. This is compounded by the complex hierarchical parallelism of modern HPC systems [148]. From a programming perspective, the question therefore is, how do we successfully exploit all this new potential parallelism without departing too radically from our current programming practices? After all, we want to able to retain existing programs and software that have proven successful to as large an extent as possible.

2.3 Distributed computing

A *distributed* system consists of many separate *nodes* connected in a network which work together so that the user can essentially consider them to form a

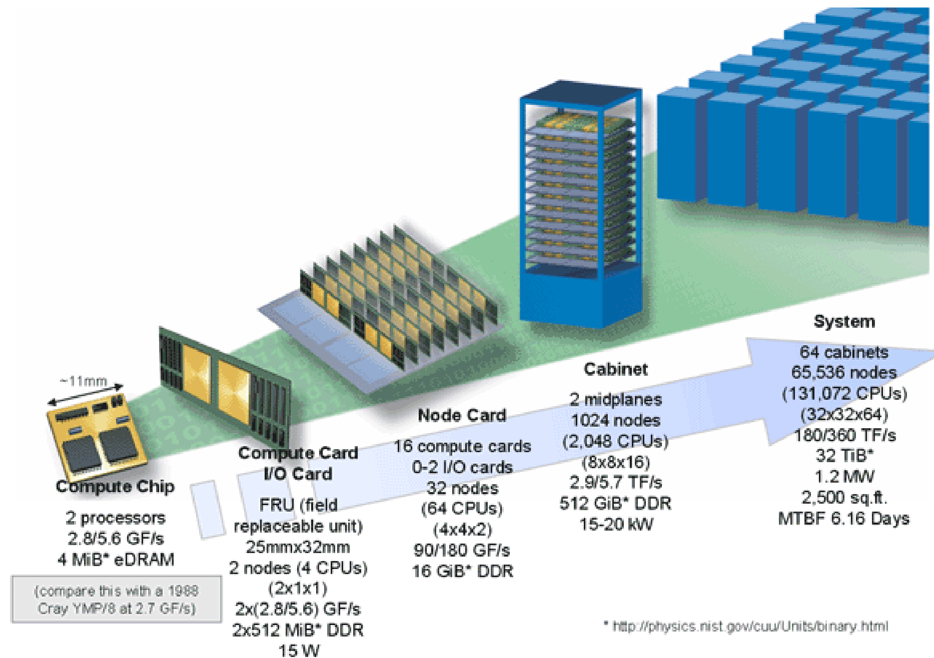

Figure 2.1: IBM's Blue Gene supercomputer architecture. Source: [154]

single system. A node may be any discrete computing unit and could be a physical hardware device or even a software abstraction (e.g., 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 [142]. 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 [11].

Distributed HPC systems can generally be divided into two types: *cluster computing* and *grid computing*. 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 [142].

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. Although we shall not explicitly discuss it

any further, cloud computing is a growth area at the moment and anyone working in HPC should at least be aware of it and its potential.

In any multiprocessor computing environment, memory management is of the utmost importance. 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 the prevailing trend at the moment is for hybrid systems where the nodes are shared memory—i.e., with all processors in the node having access to the same common memory—but memory is distributed across the system itself, with an *interconnect* used to send messages (e.g., data) between the nodes. Specialized interconnects are often used in modern HPC systems but they are still usually relatively slow so communication between processors on different nodes is often expensive.

Like parallel architectures, distributed systems have long been important in both HPC and general computing, but have increased in both prominence and complexity in recent years.

2.4 Task parallelism

One of the most popular modern parallel programming paradigms is based on the concept of a *task*—a logically discrete unit of work. The main idea is that we can simply divide up the application we wish to execute into a set of tasks, of any size or type we choose. We then identify which tasks can be done in parallel and which cannot by specifying all the dependencies between them (e.g., "we must complete task A before doing task B" or "task C needs some information from task D before it can be done"). Finally, we distribute the tasks to the available computational resources according to some scheduling strategy.

Two big advantages of the task-based programming model are as follows.

1. Extreme portability. Once we have decided how we define our tasks and specified what dependencies there are between them, we can effectively regard our application as a graph (see Section 2.4.1) and thus the application can be ported anywhere in that form.

2. Writing parallel programs becomes much easier. The programmer basically just needs to write sequential code to be executed at the task level and specify the dependencies between the tasks (e.g., how they access shared data). The tricky part is ensuring those tasks are then assigned to the available computational resources in such a way that they can be efficiently executed in parallel; this can be handled by a runtime system to further alleviate the burden on the programmer [20].

Another attraction from our perspective is that task-based programming has proven to be particularly well-suited for linear algebra applications. This goes

hand-in-hand with the recent popularity of tiled matrix algorithms. The idea is that we define our tasks to be BLAS calls (see Appendix B) on tiles of the matrix, where we choose the size of the tiles (i.e., the *granularity* of our tasks) however we see fit (for example, so they will fit into cache). This allows us to achieve our goal of exploiting more parallelism than before while still retaining existing code that has proven to be popular and effective—in this case, BLAS implementations [1]. We give one example of how this may be done for the block Cholesky factorization of a matrix in the following section.

2.4.1 From tasks to DAGs

One of the biggest advantages of task-based programming is that we can express our application as a *Task Dependency Graph* (TDG), where each node of the graph represents a task and the edges the dependencies between them. These task graphs will be *Directed Acyclic Graphs* (DAGs), i.e., directed graphs that contains no cycles. From now on, we may use the term *DAG* interchangeably with *task graph* or *TDG*.

For example, Zafari, Larsson and Tillenius in [160] give one possible task graph for the 4×4 block Cholesky factorization of a matrix A, which we reproduce in Figure 2.2. Here, A_{ij} represents the (i, j) submatrix of the matrix A and the colours of the nodes represent BLAS calls: grey = `SYRK` (symmetric rank-k update), pink = `POTRF` (Cholesky factorization), blue = `TRSM` (triangular matrix solver), and green = `GEMM` (matrix-matrix multiplication). See Appendix B for pseudocode for the block Cholesky factorization of a matrix, as well as other important definitions and concepts from numerical linear algebra.

The factorization proceeds as follows. First, we call `POTRF` to perform a Cholesky factorization of the submatrix A_{00}. We then use this to update the rest of the blocks in the leading panel using `TRSM`. Then we update the diagonal blocks with `SYRK` and the rest of the blocks below the diagonal with `GEMM`. Now we repeat the procedure beginning with A_{11}, and continue in this fashion until the entire matrix A has been factorized. The key thing to note here is that the tasks on each level of the DAG can be executed in parallel.

Apart from portability, another reason why the abstraction from applications to DAGs is so useful is that it allows us to bring known results and 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 [58]. In the case of a graph, the critical path is just the longest of all possible paths through it. This is important for us 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 application represented by that DAG.

Finding the longest path in a DAG is a well-understood problem for which

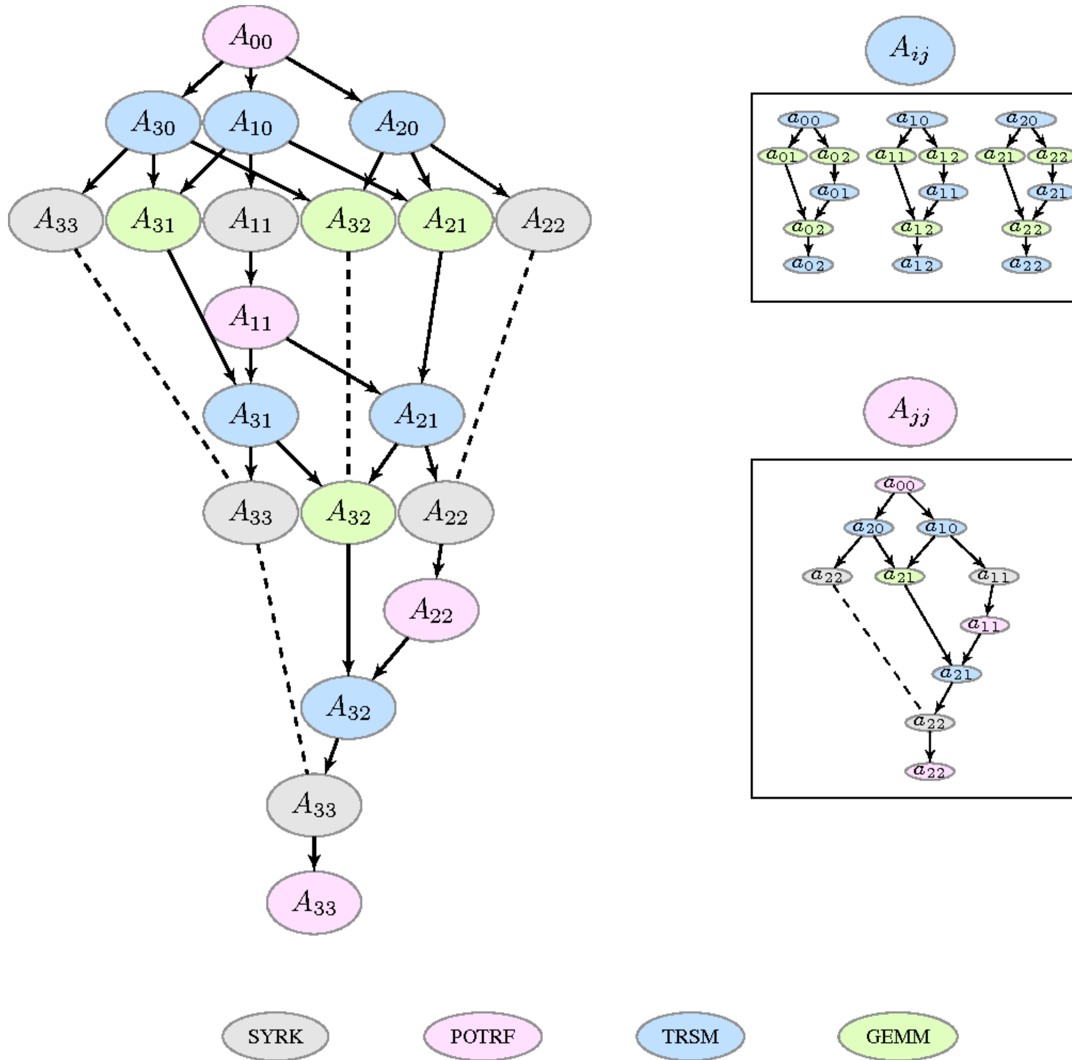

Figure 2.2: Possible task graph of a 4×4 block Cholesky factorization of a 12×12 matrix A, taken from [160]. Solid lines indicate task dependencies and dashed lines represent the fact that the tasks may be re-ordered but cannot be executed concurrently. The two bordered boxes show task graphs for block `TRSM` and `POTRF` calls on a 3×3 submatrix A_{ij}.

many methods exist for finding a solution. Of course, in practice things will be messier than we thus far supposed; for example, we will usually be dealing with *estimates* of the weights of the nodes in the graph, about which we will have some degree of uncertainty. Still, good approaches exist. The idea of planning along the critical path of an application's task graph motivates many of the scheduling algorithms that we shall see in Chapter 3.

We can distinguish two classes of task-based programming models: *Sequential Task Flow* (STF) models and *Parameterized Task Graph* (PTG) models. The former is the more common: we construct the DAG based on the order in which tasks are inserted and how data is accessed, so, in particular, we always have access to the entire DAG. In contrast, in PTG models, we never need to see the entire DAG at once. Tasks and their associated dependencies are expressed symbolically, and each node of the graph can use this symbolic representation to extract the portion of the DAG relevant to the tasks it has to execute [164]. We will see examples of software using both types of model in the next section.

2.4.2 Relevant software

Task parallelism is currently much in vogue for programming HPC systems, particularly for linear algebra [136]. OpenMP [89] has supported task-based programming since version 3.0 was released in 2008. OpenCL (*Open Computing Language*) [59] is a popular framework for parallel programming on heterogeneous platforms that allows task parallelism.

Linear algebra libraries and runtime systems that depend on the task-based approach include the following.

PLASMA (*Parallel Linear Algebra for Multicore Architectures* [37]) is a dense linear algebra software library designed for modern multicore architectures, although it only supports shared memory machines at present and is mainly intended for homogeneous systems. The PLASMA library consists of efficient implementations of parallel task-based versions of LAPACK and BLAS Level 3 routines. PLASMA routines follow the general approach for task-based linear algebra algorithms: the matrix is first converted to a tile layout and tasks are then defined at tile granularity. PLASMA was originally based on a runtime system called QUARK (*Queuing And Runtime for Kernels*), but switched to OpenMP after the latter began to support task-based programming.

DPLASMA (*Distributed Parallel Linear Algebra for Multicore Architectures* [35]) is an extension of PLASMA to distributed systems. It is very similar to PLASMA but uses a completely different task-based runtime system.

PaRSEC (*Parallel Runtime Scheduling and Execution Control* [20]) supports the scheduling of tasks in both shared and distributed memory architectures with accelerators and coprocessors. Unlike the QUARK runtime that PLASMA is based on, PaRSEC (which is used for DPLASMA) uses the PTG model.

MAGMA (*Matrix Algebra for GPU and Multicore Architectures* [36]) is an

implementation of LAPACK routines for hybrid systems consisting of multicore processors and GPUs. At present, it is limited to a single hybrid node with CPUs and a GPU but it can also support multiple GPU systems. Unlike PLASMA and DPLASMA, which are based on tile algorithms with a lot of support from runtime systems for dynamic task scheduling, MAGMA exclusively uses static task scheduling.

StarPU[4] [7] is a runtime system initially designed for heterogeneous multicore architectures and recently extended to distributed memory systems. StarPU is of particular interest to us as it allows users to implement their own custom scheduling algorithms, and implements many of the existing state-of-the-art scheduling heuristics (see Chapter 3).

StarPU supports the C programming language through extensions implemented by a GNU Compiler Collection (GCC) plug-in [28]. In Appendix C we include code for a simple implementation of the block Cholesky factorization of a matrix in this framework.

We compared the time taken for the factorization of random large (usually 4096×4096) positive semidefinite matrices with the time taken to factorize the entire matrix using the LAPACK `DPOTRF` routine (with Intel's multithreaded Math Kernel Library (MKL) [52] for optimized BLAS). All our experiments were performed on a machine with an Intel Core m3-6Y30 quad-core processor running at 0.90 Ghz with 8GB of memory; the operating system was Ubuntu 17.10. We consistently found that our StarPU implementation performed the factorization about twice as fast as using the LAPACK routine to factorize the entire matrix at once (as we would hope).

2.5 The task scheduling problem

To summarize the preceding sections of this chapter, modern HPC systems may:

1. Be highly heterogeneous, on many levels.
2. Have a hierarchy of many different levels of possible parallelism.
3. Possibly be distributed in a complex manner, for example with processors within nodes sharing memory but data being distributed across the nodes themselves.

So the problem we ultimately want to solve is: given a HPC system that may have some or all of the above features, how do we find the optimal schedule for assigning the constituent tasks of an application to the available processing units?

[4] The name is not an acronym but should be read as **PU*. Linux users should hopefully then see why this is appropriate for a runtime system targeting heterogeneous systems.

How we choose to define an optimal schedule is up to us. It could be the one which minimizes the total execution time, or the total energy consumption of the machine, or anything else we like. But whatever we choose, we should like to have an efficient algorithm which takes our system and the job we want to execute as inputs and returns an optimal schedule.

It should be clear by this juncture that our task scheduling problem is simply an instance of a more general combinatorial optimization problem (e.g., replace *processing units* with *server* and *task* with *customer* in the problem statement above), in which we have a set of jobs with various processing times and a collection of machines with different processing rates, and we want to find a schedule that optimizes some variable (usually the *makespan*, the total time it takes to process all the jobs). 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).

As a general rule, scheduling problems are usually NP-hard, except for some special cases [44]. The job shop scheduling problem with three or more machines was proven to be NP-complete by Garey in 1976 [43], as we should expect given that the traveling salesman problem is notoriously difficult. Our task scheduling problem in particular is no exception and is known to be NP-complete, even in the case where we have entirely homogeneous processors [138].

Assuming of course that P $\neq$ NP, this means that, as with all NP-complete problems, 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. Some of the most prominent examples which have been proposed are detailed in the next chapter.

We should point out at this juncture that thus far we have been assuming that we have a HPC system and have been given a specific application that we want to execute on it. Ultimately though we don't just want an algorithm that finds a good schedule for a single application—we want to be able to find good schedules for a wide range of applications, run on a wide range of system architectures. What we really want is to be able to handle arbitrary combinations of application and architecture, and still produce a good schedule. From Chapter 4 onward, we introduce a framework which we believe is a promising approach toward accomplishing this objective.

Chapter 3

Existing task scheduling heuristics

In this chapter, we give a brief overview of existing approaches to the task scheduling problem. In Section 3.1 we describe some simple scheduling heuristics that have been widely used, before moving on in Section 3.2 to discuss some of the more complex strategies that have also been proposed, including an extended analysis of *Heterogeneous Earliest Finish Time* (HEFT), which has proven to be the most popular heuristic in practice thus far.

First, we introduce some more useful concepts.

A *performance model* of a system is simply the model of the system we are using in order to make predictions about its future behavior, including all the information we have and the assumptions we are making. Clearly, the utility of a performance model depends to a great extent on its accuracy.

Load balancing means redistributing work across a network in order to improve performance (however we choose to define it, e.g., ensuring no processors remain idle or that the job is completed in the shortest possible time).

We can make a distinction between *static* and *dynamic* scheduling algorithms. 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 the execution of the job to alter the initial schedule; a simple example would be reassigning tasks waiting to be executed by one processor to another which is currently idle.

There are generic advantages and disadvantages to both approaches. We usually need a lot of information about the system (i.e., a good performance model) for a static schedule to be reliable, whereas dynamic schedules can compensate for any errors in their estimates during execution. However, dynamic schedules can often incur further computational overhead; for example, in order to reassign work to an idle processor, we first need to check all the processors to find which ones are idle, then we need to actually move the work to one of them, increasing overall communication costs.

Some consider entirely static schedulers to no longer be sufficient because of the nature of modern heterogeneous computing architectures [7, 26]. An argument could be made however that this is not true because of the growing disparity between high communication costs and cheap computation, as well as the increasing prominence of distributed memory systems, which means that dynamic schedulers may be significantly more expensive. Both of these perspectives have merit, so we may need to consider *hybrid* scheduling strategies, which mix both static and dynamic schedules. For example, we may have a distributed memory system consisting of many shared memory nodes for which communication costs between them are so expensive that we want to statically schedule tasks to the nodes but dynamically schedule the tasks within those nodes themselves.

3.1 Simple heuristics

It is not hard to think of some very simple heuristics we can use for mapping tasks to processors. The following are examples of some basic scheduling rules that have frequently been used in practice. All three are implemented in StarPU, whose naming convention we follow here.

- *Random*. Jobs are assigned to processors according to their speed, with faster processors getting more work than slower ones. We calculate (or usually estimate) the fraction of the overall performance achieved by each of the processors and then attempt to assign the corresponding fraction of the jobs to be completed to that processor. This strategy is inherently static and thus its efficiency depends to a large extent on the accuracy of the performance model of the system that is being used.

- *Work stealing*. There are two main variants of this dynamic scheduling heuristic. In the standard work stealing strategy, we assign tasks to the processors according to some other policy (for example, StarPU's implementation uses random scheduling as described above). Then once a processor becomes idle, it can take some tasks from the queue of the most loaded processor. The local work stealing strategy avoids the overhead of having to determine which processor is the most loaded by only allowing an idle processor to take a task from a neighbor (however we choose to define *neighbor*), who may not even be the most loaded neighbor.

- *Eager*. All of the tasks to be executed are stored in a single queue, shared by all the processors. Once a processor becomes idle, it retrieves another from the central queue. This is clearly a dynamic scheduling strategy.

The performance of these was investigated in [164] by Zounon[1] as part of the *Parallel Numerical Linear Algebra for Future Extreme Scale Systems* (*NLAFET*)

[1] To which this author also contributed.

project [87]. A novel linear algebra library christened *PlaStar* which builds the PLASMA library on top of the StarPU runtime system was created in order to facilitate the investigation; source code is available at the project's Github page [86]. Numerical experiments with all three heuristics (four, differentiating the standard and local work stealing scheduling strategies) were performed on three different homogeneous architectures using two tiled PLASMA algorithms (QR and Cholesky factorization). Comparisons were made with the performance achieved using PLASMA with the default OpenMP runtime and scheduler. It was found that in general the simple heuristics were fairly efficient, usually only slightly under performing relative to the OpenMP version, although the random strategy in particular sometimes performed poorly. However, these results were for homogeneous, shared memory systems; for heterogeneous, possibly distributed memory, architectures, it has been found that simple heuristics like those we have described thus far are generally inadequate [7].

3.1.1 Relating to queuing theory

We can analyze the simple heuristics given above from the perspective of queuing theory. This is not a rigorous analysis and we did not pursue this approach any further but it may still be illustrative, particularly as it may go some way towards explaining the numerical results presented by Zounon.

It is easy to see how we can view this as a queuing problem: we have a collection of processors (or *servers* in the parlance of queuing theory) and a set of tasks (or *customers*) that need to be processed (i.e., served). We wish to minimize the number of tasks still to be completed (i.e., the length of the queue). This is a basic queuing theory problem and if we make certain reasonable assumptions, a lot can be said.

The standard way to model such a system is to assume that the time between arrival of new jobs and the time it takes to process a job are independent random variables, from different distributions. We are interested in the length of the *queue*—the number of tasks currently being processed or waiting to be processed at any given time—which is a random process. In the particular case where the arrival times of jobs and the processing times of those jobs are both exponentially distributed, this random process is a continuous time *Markov chain*.

In order to relate the random scheduling heuristic as described above to a well-known queue, we first consider the case of a single processor, as that makes the exposition cleaner; we will then extend it to many. Suppose the arrival time of jobs to be processed is exponential with parameter λ and the processing time is exponential with parameter μ. Then the number of tasks still to be processed (i.e., in the queue) is a Markov chain $(X_n)_{n\geq 0}$.

This kind of queue is known as an M/M/1 queue (*memoryless inter-arrival times/memoryless service times/one server*), and it is well understood. For example, in [88] Norris shows the following:

- If $\lambda > \mu$ then $(X_n)_{n\geq 0}$ is *transient*—the length of the queue is unbounded in the long term.
- If $\lambda < \mu$ (which we shall assume from now on) then $(X_n)_{n\geq 0}$ is *positive recurrent*, i.e., the expected time between visits of all states is finite, and it has the *stationary distribution*

$$\pi_i = (1-\rho)\rho^i,$$

  i.e., the probability of the system being in state i is $(1-\rho)\rho^i$. The parameter $\rho = \lambda/\mu$ is called the *load*. Note that π_i is a geometric distribution with parameter ρ.
- When the system is in equilibrium, the average length of the queue is simply $\lambda/(\mu - \lambda)$.
- The average length of time each processor is continuously busy is $1/(\mu-\lambda)$. This ratio is also the mean waiting time for any given task to be executed.

When we have multiple processors, each of the (say, N) processors in this strategy effectively has its own queue, so extending this analysis to that case is straightforward. Ultimately, we find that the overall system load—the ratio of the total arrival rate of jobs to the total processing rate—is the sum of N geometric random variables ρ.

Similarly, the eager heuristic as described in the previous section can be related to another well-known queue. We have a single queue of tasks, shared by multiple (say, s) processors, and as soon as a processor becomes idle, the next task in the queue goes to that processor; this is a classic example of an M/M/s queue (*memoryless inter-arrival times/memoryless service times/s servers*).

Assume again that jobs arrive in the queue at rate λ. Assume also that each worker can process a job at rate μ—i.e., they are homogeneous, which was indeed the case in Zounon's numerical experiments. If i workers are busy, then the first job in the queue (which includes those currently being processed) is completed at a time which is the minimum of i independent exponential times of parameter μ. Therefore, the time taken to complete the first task is exponential with parameter $i\mu$. When all the workers are busy, the time taken to complete a task is therefore exponential with parameter $s\mu$. We see again that the queue size is also an example of a Markov chain.

This time the chain is transient if $\lambda > s\mu$. If this does not hold, then the chain is recurrent and the *detailed balance equation* gives

$$\begin{aligned} \pi_i \cdot i\mu &= \pi_{i-1}\lambda \\ \implies \pi_i &= \frac{\lambda}{i\mu}\pi_{i-1} \\ &= \frac{\rho^i}{i!}\pi_0, \end{aligned}$$

where $\rho = \lambda/\mu$ is the load. Note that in the particular case when we allow $s = \infty$, we can take $\pi_0 = e^{-\rho}$ so that

$$\pi_i = \frac{\rho^i}{i!} e^{-\rho},$$

which is a Poisson distribution with parameter ρ.

The important takeaway from this is that, as the number of workers increases, we expect the distribution of the total system load to increasingly resemble a Poisson process, which has a much narrower tail than, for example, the geometric distribution of the random scheduling strategy.

We can easily—albeit very informally—relate the local work stealing strategy to the queuing theory analysis performed for the eager strategy, as follows. Effectively, each group of "neighbors" forms a subsystem that works according to something very similar to the greedy algorithm, and therefore the number of tasks in each "subqueue" is a Poisson process by the same analysis as before. Hence if we have N groups of "neighbors" then the total system load would (roughly) be the sum of N Poisson processes.

As emphasized earlier, this is a very informal analysis in which we have made a lot of assumptions that may not actually hold. However, our analyses of the different scheduling strategies from this perspective could go some way towards explaining the behaviour observed in Zounon's numerical experiments, if we do assume that this is (at least approximately) an appropriate model. Certainly, the fact that the overall system load is geometrically distributed for the random strategy, rather than the exponential distributions of the other strategies, could explain its relatively poor performance.

Once again, we have not pursued this queuing theory type of analysis any further, but we felt it should be included here in case we decide to take this perspective again at some point in the future.

3.2 More complex scheduling strategies

Both Topcuoglu, Hariri and Wu in [138] and more recently Chronaki et al. in [26] divide all of the existing scheduling heuristics that have been proposed for both homogeneous and heterogeneous architectures into four categories: listing, clustering, guided random and duplication-based schedulers. The majority of examples in all these classes were intended for homogeneous architectures, although many of these can be easily extended to the heterogeneous case. Others were explicitly intended for heterogeneous systems.

3.2.1 Clustering schedulers

These scheduling strategies work by first clustering the tasks to be executed according to some chosen policy and then assigning each cluster to a specific

processor. The clustering step is generally done with the assumption of infinitely many processors, and the resultant clusters are then (if need be) themselves merged together to match the number of processors; Topcuoglu, Hariri and Wu believe this makes heuristic algorithms of this form impractical [138], although some of the algorithms proposed since then (2002) claim to have overcome this handicap [48].

Example heuristic algorithms in this class can be found at [48], [55], [71], [155], and [159]. Topcuoglu, Hariri and Wu note that the Levelized Min Time algorithm from [55] in particular was explicitly designed for heterogeneous architectures [138], although it performed relatively poorly in the numerical experiments they conducted with other algorithms.

3.2.2 Guided random schedulers

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 [138].

The most prominent approach of this kind for the task scheduling problem has been *evolutionary algorithms*, which take inspiration from the biological sciences. Evolutionary algorithms are an example of what is called a *metaheuristic*—basically, a heuristic for choosing heuristics in which solve a problem (i.e., we search for a good algorithm for solving the problem rather than searching for a solution directly). All evolutionary algorithms 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 to produce a new *generation* of individuals. Here, when we say "breed" we mean that we combine schedules in some biologically inspired way—for example *mutation*—in order to create new ones.

4. Repeat Steps 1–3 for the offspring, until a satisfactory individual is found.

The most widely-used kind of evolutionary algorithms are *genetic algorithms*, a term which is sometimes used as a synonym. They take inspiration in particular from how the genome of two biological parents combines in order to form offspring. We will not describe how they embellish the basic evolutionary algorithm framework given above, but the curious reader can find a good general introduction, for example, here: [152].

Evolutionary algorithms have been a very popular technique for many different scheduling problems and can often produce optimal or nearly optimal schedules in practice. However, they generally take a relatively long time to actually find a schedule compared to other methods. For example, it was found in [21] that genetic algorithms usually produced better schedules than several other scheduling heuristics, but took up to 300 times as long to do so. Topcuoglu, Hariri and Wu suggest that any performance gains achieved by finding superior schedules are usually negated by the greater time genetic algorithms need to produce a schedule [138], although this may not always be the case; see Section 9.4.3 for further discussion along these lines.

Another potential drawback when using genetic algorithms is that they normally require a lot of testing in order to find optimal values for the control parameters (e.g., mutation rate) that they use [138].

Examples of this kind of scheduler include [31], [42], [50], [57], [72], [92], [93], [95], [136], [147], [151] and [158]. Note that many of these were intended for heterogeneous systems.

Other families of metaheuristics have been applied to the task scheduling problem. One such example is what are called *chemical reaction algorithms*. These take inspiration from molecular interactions in chemistry in order to randomize their candidate solutions. Examples can be found at [69] and [157], both of which are intended for heterogeneous systems.

3.2.3 Duplication-based schedulers

Data awareness is more important than ever because of the disparity between cheap computation and expensive communication, and data duplication is a straightforward way to avoid excessive communication costs; this basic idea is widely used throughout distributed computing. The motivation behind schedulers in this class is to reduce communication costs by ensuring that successive tasks are each scheduled on the same processor, even if this requires tasks to be duplicated across different processors [26, 138]. Writing in 2002, Topcuoglu, Hariri and Wu stated that heuristics in this class were generally for an unbounded number of homogeneous processors and have high time complexity bounds relative to the other classes [138], however there are more recent strategies that are explicitly intended for heterogeneous architectures [2].

Examples of this class of schedulers can be found at [2], [4], [10], [27], [61], [96], and [163].

3.2.4 Listing schedulers

Regarded by Topcuoglu, Hariri and Wu as generally the most practical and best-performing class [138], listing schedulers work by assigning a priority to all of the tasks in the DAG according to some chosen policy (the *task prioritizing* phase)

and then assigning the tasks to the processors accordingly (the *processor selection* phase) [26, 138]. Almost all of the strategies based on the critical path approach are of this form.

As with the other classes, many of these heuristics were intended originally for homogeneous processors, but there are also some that were created for heterogeneous systems. The most prominent of these is *Heterogeneous Earliest Finish Time* (HEFT), proposed by Topcuoglu, Hariri and Wu in [138]. This is arguably the most popular of all scheduling strategies for heterogeneous systems at present. In the following section, we give a brief description of the HEFT algorithm and some later variants that have been proposed.

HEFT and its variants

Suppose we have a DAG with v tasks and e edges connecting the tasks, and we are working in a heterogeneous environment with q processors. Let w_{ij} be the estimated time it takes to execute task t_i on processor p_j (if we wish to optimize anything other than execution time, we can define w_{ij} accordingly, such as estimated power consumption if we are concerned with energy costs). Then we define the *average execution cost* of task t_i by

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

Now we give a few other definitions we need in order to state the HEFT algorithm. Let d_{ij} be the amount of data that needs to be transmitted from task t_i to task t_j. Let b_{mn} be the data transfer rate between processors p_m and p_n, and $\overline{B}$ be the average of all such transfer rates between processors. Define S_m to be the communication startup cost of processor p_m, and $\overline{S}$ to be the average of all processor communication startup costs.

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

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

and the *average communication cost* of the edge (i, j) is defined by

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

HEFT then uses the concept of the *upward rank* of a task. This is basically the length of the critical path of the DAG from that task to the end, including the cost of the task itself [138]. More formally, for any task t_i in the DAG, we define its upward rank $rank_u(t_i)$ recursively, starting from the end task, by

$$rank_u(t_i) \coloneqq \overline{w_i} + \max_{t_j \in succ(t_i)} (\overline{c_{ij}} + rank_u(t_j)),$$

where $succ(t_i)$ is the set of tasks that immediately succeed task t_i.

We conclude the task prioritizing stage of the HEFT algorithm by listing all the tasks in decreasing order of upward rank, with ties broken randomly.

In the processor selection stage, the algorithm assigns the task at the top of the list (i.e., the one with the highest upward rank) to the processor that it estimates will finish the execution of the task at the earliest possible time [26]. To make this estimation, let $AFT(t_i)$ be the actual time taken to complete task t_i and define $EST(t_i, p_m)$ to be the earliest time processor p_m can begin to execute task t_i, i.e.,

$$EST(t_i, p_m) = \max\left\{A_m, \max_{t_k \in pred(t_i)} (AFT(t_k) + c_{ki})\right\},$$

where A_m is the earliest time processor p_m will be ready to execute a task and $pred(t_i)$ is the set of tasks which immediately precede task t_i. Then the *earliest finish time* for task t_i on processor p_m is

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

With all of these definitions in place, we give a complete description of HEFT in Algorithm 3.1.

Algorithm 3.1: The HEFT algorithm.

```
1  Set the computation costs of tasks and communication costs of edges
     with mean values.
2  Compute rank_u for all tasks by traversing the DAG upward, starting
     from the end task.
3  Sort the tasks into a scheduling list by nonincreasing order of rank_u.
4  while there are unscheduled tasks in the list do
5  |   Select the first task t_i in the list.
6  |   for each processor p_k do
7  |   |   Compute EFT(t_i, p_k).
8  |   end
9  |   Assign task t_i to the processor p_m that minimizes EFT(t_i, p_m).
10 end
```

If we have q processors and e edges in our DAG, then HEFT has a time complexity of $O(p \cdot e)$. 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 $O(v^2 q)$.

Topcuoglu, Hariri and Wu performed numerical experiments to compare HEFT with three other scheduling algorithms: *Dynamic Level-Scheduling* [114], *Mapping Heuristic* [38], and *Levelized-Min Time* [55]. They primarily used two metrics to compare schedule quality:

1. *Schedule length ratio*, the ratio of the makespan of the schedule produced to the minimum possible makespan (the sum of the minimum computation times of all tasks on the critical path).

2. The *speedup* achieved—the ratio of the sequential execution time of the entire DAG and the actual makespan of the schedule produced.

They found that HEFT generally outperformed all of the three alternatives on both measures with randomly generated DAGs [138].

Introduced in the same paper as HEFT is the *Critical-Path-On-a-Processor* (CPOP) heterogeneous scheduling heuristic, which differs from the former in that both the upward and downward rank of tasks are considered, where the downward rank of a task is basically the longest path from the start to that task, excluding the cost of the task itself. At the task prioritizing stage, tasks are listed in decreasing order according to the sum of their upward and downward ranks, instead of just the upward rank [138]. The other main difference between CPOP and HEFT then comes at the processor selection stage, where CPOP tries to schedule all the tasks on the critical path to the same processor (and the others in a similar way to how it is done in HEFT). The motivation here is to try and ensure the critical path is executed as soon as possible by reducing all communications costs along it as far as possible. Topcuoglu, Hariri and Wu found that CPOP generally performed slightly more poorly than HEFT in their numerical experiments, and it has not proven to be as popular in practice.

A variant of the HEFT algorithm is proposed in [17]. The main idea is to try and improve the quality of the schedules obtained by looking further ahead than just the immediate successors or predecessors of a task. The authors perform numerical experiments with five DAGs representing real applications which suggest that, at least in some cases, this modified algorithm can produce significantly better schedules than the original. The additional overhead from the lookahead steps increase the time it takes to find a schedule, although the authors argue this is not prohibitive.

HEFT is an inherently static scheduling algorithm. A dynamic variant (called dHEFT) is proposed by Chronaki et al. in [26] and implemented in the OmpSs programming model [24]. The new algorithm assumes that we have two kinds of cores—fast and slow—and that we can comfortably divide the tasks into different types. Unlike the original algorithm, dHEFT discovers the cost of tasks at runtime, so in particular we cannot list all of the tasks in order of upward rank at any one time; tasks are instead submitted as soon as they are ready. To find the processor that will execute a task at the earliest time, for each core type we keep a record of execution times for each task type. When a task is ready, dHEFT uses these records to estimate the execution time of the task on all the processors (with exploration being performed if we do not have sufficient data to make this estimate), and schedules it on the processor which minimizes the time.

StarPU also implements a dynamic variant of the HEFT algorithm called the *deque model* (DM) scheduler which basically works in the same way as dHEFT, with tasks being scheduled on the processor with the minimal estimated execution time as soon as they are ready. Several other minor variants of this algorithm are also implemented in StarPU, such as *deque model data aware* (DMDA), which also takes data transfer times into account [122].

Although we will not do any detailed analysis here, we will make a few comments about the HEFT algorithm, with a view toward areas in which it may potentially be improved. Firstly, we can see that there is a lot of averaging being done, which although both conceptually easy and computationally cheap is not necessarily optimal. At the very least perhaps some kind of weighted averaging should be considered, although obviously we would also need to ensure that any associated overhead costs are minimized. Secondly, as previously noted, HEFT is a static algorithm. One possible approach for adapting it to a dynamic strategy has already been discussed, but others should be investigated as well. Many of the modern runtime systems we have mentioned never see the entire DAG at any one time, so being able to act without this information would be extremely desirable for a scheduling heuristic; HEFT unfortunately needs to be able to access the whole DAG in order to prioritize the tasks according to their upward rank.

Other listing schedulers

Other examples of listing schedulers can be found at [38], [48], [51], [61], [63], [68], [73], [114], [155] and [161].

Most (although not all) of the schedulers referred to so far in this section have been static. Chronaki et al. introduce a pair of dynamic and hybrid static/dynamic schedulers based on the critical path approach in [26], and evaluate their performance on a suite of real scientific applications implemented in the OmpSs framework, in a simulated environment with two kinds of cores, "fast" and "slow". Comparisons were made with a dynamic version of the HEFT algorithm called dHEFT (described above). Their results were in general promising for their proposed heuristics, although they sometimes had difficulty balancing the work load across both core types.

Daoud and Kharma introduce a scheduling heuristic specifically intended for distributed heterogeneous systems in [30]. They make use of an extension of the critical path that they call the *Longest Dynamic Critical Path* (LDCP), which, at any stage of the scheduling, is defined to be the path from any entry task to any exit task of the DAG which maximizes the sum of communication costs and computation costs over all the processors. In numerical experiments with both randomly generated DAGs and real examples representing a fast Fourier transform application, they generally found that their new heuristic outperformed HEFT in terms of producing shorter schedules.

Chapter 4

Introduction to reinforcement learning

Machine learning is the study and design of computer systems and algorithms that can learn from experience in order to improve their performance without being explicitly programmed to do so [45]. Essentially, the aim of machine learning is to extract *features* from a set of data in order to make *predictions* about future behavior. Initially an area of research within the field of artificial intelligence, the subject has grown exponentially in recent years and found an incredibly diverse range of applications. Although generally placed in computer science, machine learning relies very heavily on statistics—some consider it to ultimately be a form of applied statistics [45]—and mathematical optimization. Due to the sheer depth of current machine learning research, we shall only give a brief outline here, with a focus on the aspects most pertinent to our problem.

At least two fundamentally different kinds of machine learning can be distinguished: *supervised* and *unsupervised* learning. In the former, we are presented with a set of *training data* consisting of example inputs and their desired outputs (sometimes called *labels*) chosen by some knowledgeable *expert*. The aim is then to use this data to extrapolate some function or other mapping which is consistent with the data in the training set and which can than be used predict outputs for new input data. Unsupervised learning differs in that no such training set is provided and the goal is to discern a function or other structure from a set of raw data.

Both supervised and unsupervised learning are extremely popular methods and have been applied to a wide variety of problems. However, we shall not pursue them any further as we are instead interested in *reinforcement learning*, which can be considered either as a type of supervised learning or as a distinct kind of learning in its own right [128, p. 3].

We believe that reinforcement learning is a promising framework in which to consider the task scheduling problem in high-performance computing. Innovative methods created to deal with difficult problems in other areas have achieved

notable recent successes (see Section 8.3) and many of these may be adapted to help us find good schedules our problem. Certainly we suggest that this avenue is at least worth investigating, and this is currently the main aim of this PhD research.

In the remainder of this chapter and the four following, we provide a short introduction to reinforcement learning and its methods. The structure of our exposition closely follows the first half of *Reinforcement Learning: An Introduction* by Sutton and Barto [128], arguably the definite introductory textbook for the field as a whole, although ours is of course much more brief and informal. We also generally conform to the notation used there for consistency. Throughout these chapters we may occasional comment on how the algorithms and techniques we introduce may be applied to the task scheduling problem in particular, something we shall then elaborate in greater detail on in Chapter 9.

4.1 The reinforcement learning framework

Reinforcement learning ultimately concerns itself with how we can intelligently learn by interaction. This is something we all do every day: if we burn ourselves on a hot iron, this interaction quickly teaches us not to do it again. In fact, the modern field of reinforcement learning was inspired at least in part by pain-and-punishment, "behaviorist" theories of how animals (including humans) learn[1]. A classic example would be training a rat to complete a puzzle by giving it cheese for taking "good" actions and electric shocks for "bad" ones.

The basic set-up is that we have an *agent* which interacts with an *environment* that it can sense and influence to some extent but about which it has some degree of uncertainty. The agent receives some scalar *reward* as a result of its actions, which it seeks to maximize, and this is the only feedback it receives. Reinforcement learning differs from unsupervised learning because the agent knows that the aim is to maximize the reward, rather than finding any kind of hidden structure in the data it gathers from its interactions [128, p. 3].

Other than the agent and the environment, Sutton and Barto identify the following as the main components of a reinforcement learning system: a *policy*, a *reward signal*, a *value function*, and (optionally) a *model* of the environment. We have already discussed the concept of rewards and the reward signal is simply a scalar sent to the agent by the environment informing it how much reward it has achieved.

A policy defines how an agent behaves. Essentially, it is a map from states

[1] As an aside: although still regarded as an occasionally useful framework, this paradigm is no longer considered to be a wholly accurate model of animal cognition [97]. Indeed, some researchers believe that machine learning will soon begin to achieve diminishing returns because of this deficiency and suggest that we should study more modern theories of animal learning for inspiration on how to proceed [75]

to the actions the agent may take when it is in that state. It can take the form of a function, lookup table, or something more complex. 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). Throughout this document, we will generally use π to denote a policy rather than the mathematical constant, although this should hopefully be clear from the context.

Just like for human behavior, in reinforcement learning we need a way to weigh immediate rewards against potential long-term losses. We do this through the value 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 reward that the agent can expect to gain starting from that state, although this doesn't always have to be the case. The most important thing is that the value function encodes the relative long-term differences in reward that we can expect to achieve from different states. Note that the value function is sometimes also called the *cost* function or *cost-to-go* function. 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 $V_\pi(\mathrm{s})$ is the value of being in state s when following the policy π).

A model of the environment is basically the agent's perception of the environment and how it behaves. This allows predictions to be made—for example, a model may allow the agent to predict the reward it will receive for taking a particular action in a given state. Reinforcement learning algorithms can be either *model-based* or *model-free* [128, p. 7]. We will encounter examples of both in later chapters.

We generally consider the interaction of the agent and the environment over a discrete series of time steps, $t = 0, 1, 2, \ldots$, although extensions to the continuous case are possible (see e.g., [15]). 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 receives a reward r_{t+1} at the start of the next time step, when it also moves into a new state s_{t+1}. Assuming the agent is following the policy π, then the probability of taking action a in state s at time t is given by $\pi_t(a \mid s)$.

How we choose to define the state of the environment and the actions we may take is is clearly important. Ideally, we want the state to tell us all the information we need in order to make a decision (i.e., what action to take), but this is not always possible, either because it is computationally impractical or due to uncertainty about the environment. Ultimately, actions can be any decision we want to learn how to make and states anything we need to know in order to decide how to make them. Similarly, our 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 our decision problem.

Sutton and Barto note that this framework may not be able to describe all decision problems accurately but it has nonetheless proven to be a very useful abstraction for many practical examples [128, p. 49].

The goal of the agent is to maximize the total cumulative reward that it receives. To do this at time t, it chooses some action a_t when it is in the state s_t. But this can sometimes be tricky. If the problem is *episodic*—guaranteed to end within some finite number of steps—then the obvious way to do this would be to simply sum the rewards. However, if the problem is infinite then this sum may not converge and there would be no way to discern between actions to take at any stage. Further, even if the problem is finite, we may not want rewards to count the same at every time step. The usual way to handle this is to introduce a *discount factor* $\gamma \in [0, 1]$ such that the agent takes actions to maximize the sum of discounted cumulative rewards. Let R_t be the *expected return* after time step t. Then at time t the agent actually seeks to minimize

$$R_t \coloneqq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

Clearly, if $\gamma = 0$ then we will simply choose the action that maximizes the immediate reward at every step, and as γ approaches one we take future rewards more heavily into account. How we choose γ can often be more art than science, although some good heuristics exist. Introducing a discount factor can often make problems more tractable analytically, and we will see in coming chapters that many results only hold when we assume we are dealing with a discounted problem.

4.2 Markov decision processes

As previously mentioned, 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 *Markov*. More formally, a state $s_t = s$ is Markov if the probability that the next state is $s' = s_{t+1}$ is independent of the history of the process so far, i.e.,

$$\mathbb{P}(s_{t+1} = s' \mid s_0, a_0, r_0, s_1, a_1, \ldots s_{t-1}, a_{t+1}, r_t, s_t, a_t) = \mathbb{P}(s_{t+1} = s' \mid s_t, a_t).$$

Any reinforcement learning problem that satisfies the Markov property is called a *Markov decision process* (MDP). If the state and action spaces at every step are finite, then it is called a finite MDP. Sutton and Barto state that understanding finite MDPs is sufficient to understand 90% of modern reinforcement learning

[128, p. 60]. Throughout the rest of this document, we will almost always assume the problem we are considering is a finite MDP.

For a finite MDP in state $s_t = s$ which takes action $a_t = a$, the probability of moving into state s_{t+1} and receiving reward $r_{t+1} = r'$ is

$$p(s', r' \mid s, a) \coloneqq \mathbb{P}(s_{t+1} = s', r_{t+1} = r' \mid s_t, a_t).$$

Given this, we can compute the expected reward for a given state-action pair $r(s, a)$ from

$$r(s, a) \coloneqq \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a] = \sum_r r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a)$$

and the overall probability of moving from state s to s' using

$$p(s' \mid s, a) = \mathbb{P}(s_{t+1} = s' \mid s_t = s, a_t = a) = \sum_r p(s', r \mid s, a).$$

4.2.1 The value and action-value functions

For any Markov decision process, we can formally define the value of a state $s_t = s$ under a policy π as the expected return after starting in state s and continuing with policy π thereafter, i.e.,

$$\begin{aligned} V_\pi(s) &\coloneqq \mathbb{E}_\pi[R_t \mid s_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)\big[r + \gamma V_\pi(s')\big], \end{aligned}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π and the summation over s' and r in the final expression is just a more compact way of writing the double summation. Note that here we have introduced 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

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

Ultimately of course our goal is to find an *optimal policy* π^* such that the expected return from following that policy exceeds all others. We define the value function for this optimal policy to be

$$V^*(s) = V_{\pi^*}(s) \coloneqq \max_{\pi} \left\{ V_\pi(s) \right\} \quad \forall s \in S. \tag{4.1}$$

Similarly, the optimal action-value function is defined by

$$Q^*(s,a) = Q_{\pi^*}(s,a) \coloneqq \max_{\pi} \left\{ Q_\pi(s,a) \right\} \quad \forall s \in S. \tag{4.2}$$

Although we will not prove it here, it can be shown that, under certain reasonable assumptions, an optimal value (and action-value) function always exists, although there may be more than one optimal policy [15]. From now on, we will generally assume that this is indeed the case for the problems we are considering.

If our state space is very large, then it may not be computationally practical (or indeed possible) to store the value function (or action-value function) for all states. In this case we generally need to use an approximation; this will be discussed in greater detail in later chapters.

4.3 Balancing exploration and exploitation

An important issue underlying almost all reinforcement learning is what is called the *exploration versus exploitation*[2] problem. It is best illustrated with an example.

Suppose we have a casino that has multiple gambling machines of the kind which are referred to (sardonically) as "one-armed bandits", which each have different probabilities of winning. Our reinforcement learning agent in this case is a gambler who wants to maximize his winnings from the bandits. But he doesn't know the expected pay-off for the machines. He has to try them (i.e., explore) in order to find out. However, once he finds a good one he is unsure whether to continue playing that one (i.e., exploit it) or whether he should resume searching in order to find an even better one. Mathematically, the problem can be viewed as the gambler trying to find the global optimum and avoid getting trapped in a local one.

A generic strategy for solving this problem is what are called ϵ-*greedy* methods. The basic idea is that for some small number 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. Many of the reinforcement learning algorithms introduced in later chapters make use of simple ϵ-greedy techniques in order to try to achieve sufficient exploration.

[2] Also sometimes called *diversification versus intensification*.

An obvious problem with ϵ-greedy methods is that although taking totally random actions is conceptually simple and will allow us to eventually explore the whole state space, we don't make full use of all the information we may have. For example, returning to the bandit example, suppose we have a machine with three arms A, B and C that we have tried one thousand, ten, and one thousand times, respectively, with average pay-offs of £10, £9 and £1. If we are following an ϵ-greedy policy, then we will try arm A the vast majority of the time, and choose one of either arms B or C the rest of the time. But we have already effectively established that arm A is better than arm C; we really only want to continue trying arm B. This problem has motivated the development of more sophisticated techniques for exploration. We briefly mention two such methods here that we have considered; this is something we also hope to explore further in future research.

A general principle often applied to deal with the exploration problem is known as *optimism in the face of uncertainty*. This essentially means that, when we are operating under uncertainty, we err toward the option that what will give us the greatest expected reward. A popular method along these lines is called the *Upper Confidence Bound* (UCB) algorithm. The idea is that we calculate upper confidence bounds for the largest reward we can expect to receive for each permissible action based on the available data, and then select the action with the highest such UCB. If we let $N_t(a)$ be the number of times that action a has been tried before time t and $\tilde{V}_t(a)$ be our estimate, at that time, of the value of taking action a, then a little analysis suggests that at time t we should select the action a_t such that

$$a_t = \arg\max_a \left\{ \tilde{V}_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right\},$$

where $c > 0$ is a parameter that effectively determines how highly we prioritize exploration, with greater c values corresponding to greater exploration.

Although we won't formally derive the expression given above (see, for example, [6]), we will give some intuition. The term being square-rooted is essentially a measure of the uncertainty we have about our current estimate of the value of action a. The quantity in the maximization is therefore a kind of upper bound on the possible true value of a, with the parameter c specifying the confidence level. Each time a is taken, our uncertainty about its value decreases through the $N_t(a)$ term, which is incremented whenever a is taken. Whenever an action other than a is taken, t increases but $N_t(a)$ does not, so the uncertainty increases due to the $\ln t$ term; we use a logarithm so that the increase gets increasingly smaller but is unbounded [128, p. 37].

Numerical experiments with bandit problems generally show that UCB methods usually outperform simple ϵ-greedy methods [128, Chapter 2]. The use of UCB algorithms has mostly been considered specifically in the context of bandit

problems but has also been investigated for more general reinforcement learning [56].

Thompson sampling or *posterior sampling* is an alternative approach for balancing exploration and exploitation that has become popular in recent years, although it actually dates back to the 1930s [105, 134, 135]. The basic idea here is that we use Bayesian inference in order to determine the best action to take at every step. In particular, we assume the we know the initial distribution of action values and then select the action which is the best according to their posterior distribution. We update the distribution after every time step. These updates can often be prohibitively complex, although if we assume we have a *conjugate prior* then they become quite simple (the beta distribution is a popular choice of prior for this reason). Empirically, Thompson sampling methods have generally been found to perform at least as well as all the other alternatives [128, p. 44], although historically issues with determining practical episode lengths have hindered their application to reinforcement learning problems. Recent research has focused on overcoming this handicap, and there are promising published results [90].

4.4 Credit assignment

A fundamental problem throughout all of machine learning, identified at least as far back as 1961 by Minsky [77] is what is called the *credit assignment* problem. The issue ultimately boils down to, how do we identify those useful features that are genuinely useful for making predictions from others that aren't? For example, suppose we have an agent that is attempting to learn to forecast the weather. Suppose also that for the first few weeks of its learning, it happens to rain on a Thursday, purely due to chance. How do we ensure that the agent does not predict rain on the following Thursday simply because this has always been the case so far?

In reinforcement learning in particular, we also have the *temporal* or *long-term* credit assignment problem. This refers to the fact that rewards for actions may not be immediately apparent; an agent may make a series of actions for which it receives negligible immediate rewards before receiving a large one, and then be unable to determine which of the preceding actions was responsible for it.

Ultimately credit assignment is in some sense linked to successfully discerning correlation and causation; to overcome it, we need to be able to distinguish the factors that are, and are not, important for predicting future rewards. The key is to make sure that how we choose to define the state, actions and rewards of our environment encodes genuinely useful information. In our weather forecasting example, the name of the day is not useful when determining whether it will or will not rain so should not be chosen as a component of the state.

We will not focus on the credit assignment problem to the same extent that we will do for the exploration and exploitation trade-off but it is something that

we must always be mindful of and attempt to mitigate.

Chapter 5

Dynamic programming

Optimal control theory concerns itself with finding the optimal way to act (the optimal *control*) in a dynamic system in order to meet some chosen criteria; all of reinforcement learning as we have presented it can thus be regarded as attempting to find the optimal control of a Markov decision process with unknown dynamics. MDPs are of course stochastic processes and, in general, the most powerful approach for solving stochastic optimal control problems is *dynamic programming*, a family of algorithms which were first introduced in the 1950s by Bellman [12]. They are distinguished from other methods principally by their use of value functions to guide the search for good policies; reinforcement learning and dynamic programming are thus in some very real sense almost equivalent [128, p. 79]. The difference is that in classical dynamic programming we assume that the dynamics of the environment are known, whereas in reinforcement learning the goal is to learn them through *experience*, so the optimal value functions and policies we achieve are estimates—hence, reinforcement learning is also often known as *approximate dynamic programming*.

(We note here that the reader should beware that—as is to be expected with the cross-disciplinary nature of reinforcement learning—different names may often be used to refer to the same thing; generally, researchers working in control theory may prefer the term *approximate dynamic programming* rather than *reinforcement learning* to refer to the field as a whole [15, p. 53], while those working in artificial intelligence almost unanimously use the latter. Similarly, the *cost* function or *cost-to-go* function may be preferred to *value* function, depending on whether the aim of the kind of problems typically encountered is to minimize costs or maximize rewards.)

The guiding principle in dynamic programming is simple: if we are in a particular state and we have a policy that is optimal, then it remains optimal if we start from the next state instead of the current one. In particular, if we are at time t, we take the action which optimizes the sum of the immediate reward at this time step and the reward that we expect to receive by acting optimally from time $t+1$ onward; this procedure of updating estimates based on other estimates

is known as *bootstrapping* and is one of the defining characteristics of dynamic programming. We then solve the recursive relationship between succeeding time intervals in order to determine an optimal policy; in the finite time case, this can be done by simply working backward from the final time step.

Dynamic programming is an extremely rich subject which we cannot possibly hope to capture entirely here, so we will focus only on the aspects that are most immediately relevant for our purposes. We may also occasionally be somewhat informal; for a rigorous treatment of the subject which covers everything we introduce here in much greater detail, the reader is directed to [15]. The structure and notation of this chapter closely follow [128, Chapter 4].

5.1 The Bellman equation

Dynamic programming algorithms attempt to find an optimal policy by recursively solving the *Bellman equation*. Suppose we are in a state s_t at time t and let $\mathcal{P}_t$ be the set of all policies that can be implemented from time t onward. Any policy $\pi \in \mathcal{P}_t$ can be written as $\pi = (a_t, \Pi_{t+1})$, where a_t is the action taken at time t and Π_{t+1} belongs to the set $\mathcal{P}_{t+1}$ of all policies that can be taken from time $t+1$ onward. So we have

$$\begin{aligned}
V^*(s_t) &= \max_{\pi \in \mathcal{P}_t} \big\{ V_\pi(s_t) \big\} \\
&= \max_{\pi \in \mathcal{P}_t} \big\{ \mathbb{E}_\pi[R_t \mid s_t = s] \big\} \\
&= \max_{\pi \in \mathcal{P}_t} \left\{ \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s \right] \right\} \\
&= \max_{a_t} \left\{ \max_{\Pi_{t+1} \in \mathcal{P}_{t+1}} \left\{ \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s \right] \right\} \right\} \\
&= \max_{a_t} \left\{ \max_{\Pi_{t+1} \in \mathcal{P}_{t+1}} \left\{ r_t + \mathbb{E}_{a_t} \left[\mathbb{E}_{\Pi_{t+1}} \left[\sum_{k=0}^{\infty} \gamma^{k+1} r_{t+k+1} \;\middle|\; s_t = s \right] \right] \right\} \right\} \\
&= \max_{a_t} \left\{ r_t + \gamma \mathbb{E} \left[\max_{\Pi_{t+1} \in \mathcal{P}_{t+1}} \left\{ \mathbb{E}_{\Pi_{t+1}} \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \;\middle|\; s_t = s \right] \right\} \right] \right\} \\
&= \max_{a_t} \big\{ r_t + \gamma \mathbb{E}[V^*(s_{t+1})] \big\},
\end{aligned}$$

where the last equality is the Bellman equation. Although we will not show it here, the converse also holds—any value function which satisfies the Bellman equation is optimal [15]. So finding V^* reduces to finding a solution to the Bellman equation.

Once we have the optimal value function V^*, it is then a simple task to recover an optimal policy π^*: since we are assuming a finite MDP, at every state we just select the action that gives us the maximum value at that stage, according to V^*.

Similarly, for the optimal action-value function we have the equivalent Bellman equation

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

Some of the algorithms we shall introduce later are described in terms of the value function and others the action-value function; depending on the problem, one can often prove more convenient to work with than the other.

5.2 Policy evaluation and improvement

At a high level, many dynamic programming methods have a simple two-step structure: start with some random policy, evaluate it, find some way to improve it, and then repeat the previous two steps for the improved policy until we find one that is (sufficiently close to) optimal.

Unfortunately, evaluating the performance of a policy may not be straightforward in practice. For every state s, we must compute

$$\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_t + \sum_{k=1}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s\right] \\
&= \mathbb{E}_\pi\big[r_t + \gamma V_\pi(s_{t+1}) \mid s_t = s\big] \\
&= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V_\pi(s')\big].
\end{aligned}$$

or

$$\begin{aligned}
Q_\pi(s, a) &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s, a_t = a\right] \\
&= \mathbb{E}_\pi\left[r_t + \sum_{k=1}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s, a_t = a\right] \\
&= \mathbb{E}_\pi\big[r_t + \gamma V_\pi(s_{t+1}) \;\big|\; s_t = s, a_t = a\big] \\
&= \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V_\pi(s')\big].
\end{aligned}$$

Assuming (at least for the moment) that we know all the probabilities $\pi(a \mid s)$ and $p(s', r \mid s, a)$, then we just have a system of equations—one for each state s—that we can solve in any standard way. However, if we have many states then this may not be computationally practical.

An alternative is to use *iterative policy evaluation*. Start with any value function V_0, which may be entirely random except that the value of the terminal state—the state after reaching which the process terminates—must be zero, if it exists. Then, for all states s, we update the value function using

$$\begin{aligned} V_{k+1}(s) &= \mathbb{E}_\pi\big[r_t + \gamma V_k(s_{t+1}) \mid s_t = s\big] \\ &= \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V_k(s')\big]. \end{aligned}$$

Although it can be shown that $V_k \to V_\pi$ as $k \to \infty$ (see e.g., [15]), this is obviously not practical so we typically stop iterating when the maximum difference between successive iterates over all states goes beneath some small specified tolerance.

Now that we can efficiently evaluate the value of a policy, the next step in the rubric outlined at the beginning of this section is to find some way to improve it. Following the lead of Sutton and Barto in [128], we will consider just deterministic policies for the remainder of this section as the exposition is cleaner; the extension to stochastic policies is relatively straightforward and can be found in [15], among others.

The key way to approach the problem is to assume we are following a policy π and consider when it is better to take another action in a state s rather than the prescribed action $\pi(s)$. Suppose we have another policy π' which is identical to π except that it takes action $\pi'(s)$ rather than $\pi(s)$. The value of following the modified policy is

$$\begin{aligned} Q_\pi(s, \pi'(s)) &= \mathbb{E}_{\pi'}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s\right] \\ &= \mathbb{E}_{\pi'}\big[r_t + \gamma V_\pi(s_{t+1}) \;\big|\; s_t = s\big], \end{aligned}$$

so we want to follow the modified policy π' if this exceeds the value function $V_\pi(s)$ for all states s. Assume that this is the case. Then we have

$$\begin{aligned} V_\pi(s) &\le Q_\pi(s, \pi'(s)) \\ &= \mathbb{E}_{\pi'}\big[r_t + \gamma V_\pi(s_{t+1}) \;\big|\; s_t = s\big] \\ &\le \mathbb{E}_{\pi'}\big[r_t + \gamma Q_\pi(s_{t+1}, \pi'(s_{t+1})) \;\big|\; s_t = s\big] \\ &= \mathbb{E}_{\pi'}\big[r_t + \gamma \mathbb{E}_{\pi'}\big[r_{t+1} + \gamma V_\pi(s_{t+2})\big] \;\big|\; s_t = s\big] \\ &\le \mathbb{E}_{\pi'}\big[r_t + \gamma r_{t+1} + \gamma^2 V_\pi(s_{t+2}) \;\big|\; s_t = s\big] \\ &\le \mathbb{E}_{\pi'}\big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V_\pi(s_{t+3}) \;\big|\; s_t = s\big] \\ &\;\;\vdots \\ &\le \mathbb{E}_{\pi'}\big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \gamma^4 r_{t+4} + \dots \;\big|\; s_t = s\big] \\ &= V_{\pi'}(s). \end{aligned}$$

Thus $V_\pi(s) \leq V_{\pi'}(s)$ for all states s and therefore π' is a better policy than π. So the natural step to constructing an improved policy π' from any policy π is to make it *greedy* with respect to the value function of π, i.e., to define, for all states s,

$$\begin{aligned}
\pi'(s) &\coloneqq \arg\max_a \{Q_\pi(s,a)\} \\
&= \arg\max_a \{\mathbb{E}\big[r_t + \gamma V_\pi(s_{t+1}) \bigm| s_t = s, a_t = a\big]\} \\
&= \arg\max_a \Big\{ \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V_\pi(s')\big] \Big\}.
\end{aligned}$$

If there is more than action which achieves this maximum, then we can break the tie arbitrarily; precisely which of the optimal actions is chosen makes no difference.

So we have found a procedure that will allow us take any policy and find another policy that is at least as good. It is easy to see how we know when we actually achieve the optimal policy. Suppose we are following a policy π and our improvement step gives us a policy π' that does not strictly improve on π, i.e., $V_\pi(s) = V_{\pi'}(s)$ for all s. Then

$$V_{\pi'}(s) = \max_a \Big\{\mathbb{E}\big[r_t + \gamma V_{\pi'}(s_{t+1}) \bigm| s_t = s, a_t = a\big]\Big\}.$$

In other words, the policy π' obeys the Bellman equation and is therefore an optimal policy!

5.3 Policy iteration

The policy evaluation and policy improvement steps outlined in Section 5.2 suggest a simple algorithm which is known as *policy iteration*: start with a (possibly random) policy, evaluate it, improve it using the policy improvement procedure, and repeat until the policy does not change from one iteration to the next; this is the optimal policy. A complete description is given by Algorithm 5.1, which is a modified version of that given in [128, p. 87].

Our analysis assures us that policy iteration will eventually converge to an optimal policy, but says nothing about how long this will take. Examples can be constructed which require a number of iterations roughly equal to the number of states [109], which may of course be very large, but in practice policy iteration is usually found to converge very rapidly, often in a mere handful of iterations [128, p. 87]. Algorithm 5.1 terminates when the maximum difference between successive iterates is smaller than some specified tolerance.

Algorithm 5.1: Policy iteration, with iterative policy evaluation.

Input : Arbitrary value function V and policy π. Tolerance level ϵ.

Output: Approximate optimal value function $V \approx V^*$ and policy $\pi \approx \pi^*$.

```
   /* Iterative policy evaluation                                  */
 1 Δ = ∞
 2 while Δ > ϵ do
 3 |   Δ = 0
 4 |   foreach state s do
 5 |   |   v = V(s)
 6 |   |   V(s) = \sum_{s',r} p(s', r | s, π(s)) [r + γV_π(s')]
 7 |   |   Δ = max{Δ, |v − V(s)|}
 8 |   end
 9 end
   /* Policy improvement                                           */
10 stable = true
11 foreach state s do
12 |   old action = π(s)
13 |   π(s) = argmax_a { \sum_{s',r} p(s', r | s, a) [r + γV_π(s')] }
14 |   if old action ≠ π(s) then
15 |   |   stable = false
16 |   end
17 end
   /* Check if we are done                                         */
18 if stable then
19 |   return V and π
20 else
21 |   go to step 1
22 end
```

5.4 Value iteration

Although policy iteration typically requires few iterations, those iterations can be expensive due to the policy evaluation step, which usually involves either solving a large system of equations or performing iterative policy evaluation. But if we are using iterative policy evaluation, we don't need to wait until it has actually converged before performing the improvement step. An alternative dynamic programming algorithm known as *value iteration* combines a single step of policy evaluation and policy improvement at every iteration. In particular, we update our value function at every iteration using the formula

$$\begin{aligned} V_{k+1}(s) &:= \max_a \Big\{ \mathbb{E}[r_t + \gamma V_k(s_{t+1}) \mid s_t = s, a_t = a] \Big\} \\ &= \max_a \Big\{ \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V_k(s') \big] \Big\}. \end{aligned}$$

A fuller description of value iteration is given in Algorithm 5.2, which is a modified version of that given in [128, p. 90].

Algorithm 5.2: Value iteration.

Input : Arbitrary value function V, tolerance level ϵ.
Output: Approximate optimal value function $V \approx V^*$ and policy $\pi \approx \pi^*$.

```
   /* Find the (approximate) optimal value function                    */
1  Δ = ∞
2  while Δ > ϵ do
3  |   Δ = 0
4  |   foreach state s do
5  |   |   v = V(s)
6  |   |   V(s) = max_a { Σ_{s',r} p(s', r | s, a)[r + γV_k(s')] }
7  |   |   Δ = max{Δ, |v − V(s)|}
8  |   end
9  end
   /* Compute the (approximate) optimal policy                         */
10 foreach state s do
11 |   π(s) = arg max_a { Σ_{s',r} p(s', r | s, a)[r + γV_k(s')] }
12 end
```

Like policy iteration, value iteration has been proven to converge to the optimal value function in the limit [15]. However, unlike policy iteration, it isn't quite clear when we should actually stop iterating in practice. Typically, the same stopping criterion is used as for policy iteration—checking that successive iterates are sufficiently similar. Bounds which give stronger convergence results

have been established but it's usually unclear whether any improvement in the algorithm is worth the additional overhead of computing the bounds themselves [15, p. 245]. Value iteration typically requires more iterations for convergence than policy iteration but the iterations themselves are usually cheaper (as we would expect).

From a computational perspective, we can see that both value and policy iteration require a full *sweep* across the entire state space, which may be problematic if we have a large number of states. Both value and policy iteration are widely used, and the ideas behind both form the basis for many of the more complex reinforcement learning algorithms described in later chapters.

5.5 The curse of dimensionality

It is often said that dynamic programming suffers from the *curse of dimensionality*, a term used to describe the difficulties that can arise because the state space grows exponentially larger as the number of state variables increases; we saw for example that both value and policy iteration require searching the entire state space, which may be practically impossible if the total number of states is exponentially large. Despite this, Sutton and Barto suggest that, although theoretical complexity bounds may be higher, dynamic programming actually handles large state spaces relatively well compared to alternative methods, such as linear programming, at least in practice [128, p. 94].

Still, large state spaces can be awkward, particularly at the very largest scale, when it becomes computationally impractical to store the values of all states. However, clever methods have been devised to extend dynamic programming techniques to many of these problems. One such approach works by approximating the value (or action-value) function and so is known as *approximate dynamic programming*; this is discussed further in Chapter 8.

Chapter 6

Monte Carlo learning

In this chapter we discuss one type of reinforcement learning methods and the general theory behind them. Fundamentally, they all follow the general framework introduced in Section 5.2, which is sometimes known as *generalized policy iteration*: we have policy evaluation and policy improvement steps, and these interact in some—possibly very complex—manner in order to converge to an optimal policy.

In reinforcement learning we do not have perfect knowledge of the environment so we must learn estimates for the system's dynamics (in particular, optimal policies and value functions) through experience. In practice, it is very useful if we can learn from *simulated* experience. There are problems for which gaining real experience is difficult or impossible; robotics is a good example, although we can also see that learning from simulated experience would be very helpful for our task scheduling problem in HPC as well due to the costs associated with running such systems (see Section 9.2). Of course, the problem with learning from simulated experience is that the simulator needs to be reliable, and no matter how good it is we have introduced another element of approximation into matters. Still, the use of simulation for reinforcement learning is well-established and has a solid theoretical foundation [16].

Similar to the previous chapter, the structure and notation used in this chapter closely follows that of [128, Chapter 5].

6.1 On-policy and off-policy methods

We gain experience in order to learn optimal value functions and policies, but in order to actually gain this experience we must follow some policy or policies. We can make a fundamental distinction between at least two classes of reinforcement learning algorithms. *On-policy* methods attempt to balance the goal of attaining an optimal policy with fully exploring the state space and, rather than finding a truly optimal policy, instead settle for finding a nearly optimal one that also

explores. The other class of methods are called *off-policy* methods and they try to achieve the same balance by using *two* policies, one that is updated toward the optimal policy (the *target policy*) and another that is used for exploration in order to gather data (the *behavior policy*). Note that on-policy methods can thus be considered a kind of off-policy learning in which the target and behavior policies are the same, so they are more general in this sense.

On-policy methods are usually the more straightforward of the two because off-policy methods have to account for the *prediction problem*—the fact that the data used to update the target policy comes instead from the behavior policy. Indeed, it is not immediately obvious that this can be done at all, but in fact we shall see that it can be if certain reasonable assumptions are made. Off-policy methods tend to have greater variance than on-policy methods for this reason, but because of their greater generality are potentially much the more powerful of the two.

Suppose that π is our target policy and we are following the behavior policy μ, where both are fixed. In order for data gathered from following μ to be applicable to π, we require what is called *coverage*: all actions taken by π must also be taken by μ at least once. More formally, coverage means that

$$\pi(a \mid s) > 0 \implies \mu(a \mid s) > 0, \quad \text{for all states } s \text{ and actions } a \in \mathcal{A}(s).$$

If we make the assumption that μ covers π, then it is clear that μ must be stochastic for those states for which it is not identical to π; however, π itself can be deterministic. In fact, we shall see this is often the case; typically, π may be the deterministic policy which is greedy with respect to the current estimate of the value function (although here we are assuming that it is fixed).

A useful concept for off-policy methods is *importance sampling*. This is a general probabilistic technique for estimating expected values from one distribution using samples from another. Suppose we are in state $s = s_t$ and consider the *trajectory* we follow from time t onward, $s_t, a_t, s_{t+1}, a_{t+1}, \ldots, s_{T-1}, a_{T-1}, s_T$. Then the basic idea behind using importance sampling for off-policy methods is that we weight the expected return of a trajectory using the *importance-sampling ratio*, the relative probability of the trajectory occurring under both policies. In particular, the probability of the given trajectory occurring under any policy π is

$$\prod_{k=t}^{T-1} \pi(a_k \mid s_k) p(s_{k+1} \mid s_k, a_k),$$

where $p(s_{k+1} \mid s_k, a_k)$ is the total probability of moving into state s_{k+1} after taking action a_k when in state s_k. Then we define the importance-sampling ratio for that trajectory (for the target and behavior policies) to be

$$\rho_t^T := \frac{\prod_{k=t}^{T-1} \pi(a_k \mid s_k) p(s_{k+1} \mid s_k, a_k)}{\prod_{k=t}^{T-1} \mu(a_k \mid s_k) p(s_{k+1} \mid s_k, a_k)} = \prod_{k=t}^{T-1} \frac{\pi(a_k \mid s_k)}{\mu(a_k \mid s_k)}, \tag{6.1}$$

where we can see that the transition probabilities all cancel, meaning that the importance-sampling ratio depends only on the policies and not the dynamics of the MDP itself. Note that here we have assumed that we have a finite-time MDP; the extension to the infinite-time case is slightly more complicated but can also be done.

6.2 Monte Carlo algorithms

The simplest way to learn from experience is by simply taking the (possibly weighted) average of many different samples; given enough samples, this should converge to the true value. This is the basic idea behind the algorithms in this section, and all other such *Monte Carlo* methods. More specifically, we can estimate the return of a policy by averaging over many *episodes* of experience in which we follow that policy. In the rest of this section, we will assume that all such episodes are finite and eventually terminate in one way or another as the alternative is obviously not practical. Note also that we follow the lead of Sutton and Barto in using the term *Monte Carlo* to refer only to methods that work by averaging complete episode returns [128, p. 99], so in particular this means that we can update our estimated value functions and policies at the end of each episode but not during the episode itself. So Monte Carlo methods are in some sense online across episodes but not within them.

One potential issue we have is that the return from taking an action in one state depends on the actions taken later in the episode. Thus the problem is now nonstationary from the point of view of the state since the values of future actions may be updated later. To deal with this, we modify the generalized policy iteration framework. Now, we must learn value functions and policies from sample returns with the MDP, rather than just being given them as in classical dynamic programming.

Given a policy, the value of a state under that policy is just the expected return starting from that state. So the basic Monte Carlo approach to estimating $V_\pi(s)$ for a state s and a policy π is that we consider a bunch of episodes in which the environment is in state s at some point; every time the environment is in state s, we call that a *visit*. Note that in general the state s may be visited more than once per episode. This gives rise to two minor variants of the basic approach. In the *first-visit* Monte Carlo method, we estimate $V_\pi(s)$ as the average return following the first time we visit s; on the other hand, the *every-visit* method averages the returns following all visits to s.

It can be shown that both variants of the Monte Carlo method converge to the true value of $V_\pi(s)$ as the number of visits or first-visits (respectively) tends to infinity. In the first-visit method, each return is an independent, identically distributed estimate of $V_\pi(s)$ with finite variance. Thus the law of large numbers tells us that the averages of the sample estimates converge to their expected

value. Each average is an unbiased estimate and the standard deviation of its error converges quadratically (with respect to the number of returns averaged). The proof for every-visit Monte Carlo is more complex but it can also be shown to converge quadratically [128, p. 101].

It is worth noting that even in situations in which classical dynamic programming methods can in theory be applied, Monte Carlo methods may well be easier in practice, such as when calculating the transition probabilities of the system is complex and potentially error-prone.

6.2.1 Exploring starts

If a model of the system is not available, then it is more useful to be able to estimate action values rather than state values. With a model, state values are all that is needed to find an optimal policy: we just do a one-step lookahead to find the best action at every step, as in dynamic programming. Without a model, we have to estimate the value of each action. To estimate $Q_\pi(s, a)$ for a given state s and action a we can use the Monte Carlo method already described (either first- or every-visit).

The problem now is that not every state-action pair may be visited. In particular, if π is a deterministic policy, then it will choose the same action every time it encounters a state and never explore the others; thus there will be no returns to average for the other actions and the estimates of their value will never be updated. But for policy evaluation to work for action values we need to ensure that we explore these alternative actions. One way to do this is by specifying that all episodes start in a state-action pair and that each pair has a nonzero probability of being selected; this is called *exploring starts*. For an infinite number of episodes, this guarantees that all pairs will be visited an infinite number of times.

An alternative approach that doesn't require exploring starts is to just consider policies which are entirely stochastic, with a strictly positive probability of taking all actions in every state; we will return to this later.

We now consider the use of Monte Carlo methods to approximate optimal policies. The idea is to use the generalized policy iteration framework introduced earlier: we maintain an approximate value function and an approximate policy, and repeatedly update the former to more closely approximate the value function for the current policy, and repeatedly improve the latter with respect to the current value function. In tandem, this means that both the policy and value function will eventually become optimal. Assuming (for the moment) that we do have infinitely many episodes and they are all generated with exploring starts, then it can be shown that Monte Carlo methods will compute $Q_{\pi_k}(s, a)$ exactly for all policies π_k [128, p. 106].

Policy evaluation in this framework is done exactly as described previously, using either the first-visit or every-visit variant. Policy improvement is done by

choosing the next policy π_{k+1} to be the greedy policy with respect to Q_{π_k}. Then since, for all states s,

$$\begin{aligned} Q_{\pi_k}(s, \pi_{k+1}(s)) &= Q_{\pi_k}(s, \arg\max_a Q_{\pi_k}(s,a)) \\ &= \max_a Q_{\pi_k}(s,a) \\ &\geq Q_{\pi_k}(s, \pi_k(s)) \\ &\geq V_{\pi_k}(s). \end{aligned}$$

this produces a sequence of non-decreasing policies, and therefore they converge to the optimal policy and value function. So Monte Carlo methods can find optimal policies given only sample episodes and with no knowledge of the dynamics of the system.

We have assumed that we have an infinite number of episodes, which obviously isn't practical. There are typically two ways to overcome this restriction. One way is to stick with the idea of approximating q_{π_k} in each policy evaluation. We can then then obtain bounds on the magnitude of the errors and take enough steps during each policy evaluation so that these bounds are acceptably small. Unfortunately, the nature of the bounds is such that this would generally require an unacceptably large number of steps and Sutton and Barto regard it as impractical for anything but very small problems [128, p. 107].

The second approach for avoiding the necessity of infinitely many episodes is to forgo doing a complete policy evaluation before improving the policy. The idea is that on each evaluation, we move the policy toward Q_{π_k} but do not expect to actually get there. An extreme example of this is value iteration, which does a policy improvement, then a single iteration of iterative policy evaluation, before improving the policy again.

Taking all this together suggests a good approach would be to both evaluate and improve the policy at the end of every episode; first we use the observed returns for policy evaluation, and then we improve the policy for all the states visited in the episode. Algorithm 6.1 fully describes an algorithm of this kind, a slightly modified version of that given in [128, p. 107]. Note that all returns are averaged for each state-action pair without any consideration as to what policy was being followed when they were observed. Although infinitely many episodes are required for convergence in theory, we stop the iterations when differences between successive iterates is sufficiently small, as we did for the value and policy iteration algorithms given in the previous chapter.

Intuitively, it seems clear that Monte Carlo method with exploring starts must converge to an optimal policy since it only achieves stability when both the policy and value function are optimal and changes to the action-value function decrease over time, however this has not been proven; Sutton and Barton consider this to be one the most important open problems in reinforcement learning, although a partial solution is given in [141].

Algorithm 6.1: Monte Carlo control with exploring starts.

```
/* Inputs and outputs */
```

Input : Small tolerance ϵ. Action-value function $Q(s, a)$ and policy $\pi(s)$, arbitrary for all states s and actions $a \in \mathcal{A}(s)$. R(s, a), array of arrays for storing the observed returns for all state-action pairs.

Output: Updated policy $\pi \approx \pi^*$ and action-value function $Q \approx Q^*$.

```
/* Start of algorithm proper */
```

1 $\Delta = \infty$
2 **while** $\Delta > \epsilon$ **do**
3     $\Delta = 0$
4   Choose an initial state s_0 and action $a \in \mathcal{A}(s_0)$ at random.
5   Generate an episode by starting from s_0, taking action a_0, then following π thereafter.
6   **foreach** *pair* (s, a) *appearing in the episode* **do**
7   r = return following the first-visit to (s, a)
8   Append r to R(s, a)
9         $Q(s, a)$ = average of all entries in array R(s, a)
10   **end**
11   **foreach** *state* s *in the episode* **do**
12         $p = \pi(s)$
13         $\pi(s) = \arg\max_a Q(s, a)$
14         $\Delta = \max(\Delta, |p - \pi(s)|)$
15   **end**
16 **end**

6.2.2 Without exploring starts

To alleviate the necessity for exploring starts while remaining with an on-policy method, we can use a *soft* policy, i.e., a stochastic policy π such that $\pi(a \mid s) > 0$ for all states s and actions $a \in \mathcal{A}(s)$, which slowly moves toward a deterministic optimal policy. We have already seen an example in the previous chapter of a soft policy in the ϵ-greedy method for ensuring continued exploration, where most of the time we choose an action that has the optimal expected value, but with probability ϵ we select another action at random; more formally,

$$\pi(a \mid s) = \begin{cases} 1 - \epsilon - \frac{\epsilon}{|\mathcal{A}(s)|}, & \text{if } a \text{ is optimal,} \\ \frac{\epsilon}{|\mathcal{A}(s)|}, & \text{otherwise.} \end{cases}$$

We define a policy to be ϵ-*soft* if $\pi(a \mid s) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$ for all states and actions, so ϵ-greedy policies are an example of ϵ-soft policies.

To now define an on-policy Monte Carlo algorithm for control that doesn't require exploring starts we work once more in the framework of generalized policy iteration. To begin, we use first-visit Monte Carlo sampling to estimate the action-values for our current policy. Then this time we move the policy toward a policy which is ϵ-greedy; we can do this since generalized policy iteration doesn't actually require that we move all the way to a greedy policy, just toward it. It can be shown that for any ϵ-soft policy π, any ϵ-greedy policy with respect to Q_π is guaranteed to be better than or equal to π, although we will not do so here; see [128, p. 110]. Therefore, as before, stability is only achieved when we have an optimal value function, so policy iteration also holds for ϵ-soft policies. Ultimately this means that we don't achieve the optimal policy among all policies, but only the the optimal ϵ-soft policy, but this will often suffice.

Algorithm 6.2 is a modified version of that given in [128, p. 109], which follows the above procedure. Note that here we terminate the `while` loop when an optimal policy is found, which can be established by checking if it has changed relative to the previous one, although this isn't shown in our pseudocode as it made it untidy. In practice, a maximum number of episodes is usually also set.

6.2.3 Off-policy Monte Carlo methods

To completely remove the dependence on exploring starts, we need to move to off-policy methods. The key is to use the idea of importance sampling introduced previously. First of all, we need to be able to estimate $V_\pi(s)$ for a policy π and state s from a batch of episodes following a different policy μ. It is convenient here to number all time steps incrementally, even across episodes—i.e., if episode 1 ends at time step 100, then episode 2 starts at time step 101. This way we can distinguish between time steps in particular episodes. So in particular we can define the set $\mathcal{T}(s)$, which is the set of all time steps in which the state s is

Algorithm 6.2: First-visit Monte Carlo control for ϵ-soft policies.

```
/* Inputs and outputs */
```

Input : Constant ϵ. Action-value function $Q(s, a)$ and ϵ-soft policy $\pi(a \mid s)$, arbitrary for all states s and actions $a \in \mathcal{A}(s)$. R(s, a), array of arrays for storing the observed returns for all state-action pairs.

Output: Updated ϵ-soft policy $\pi \approx \pi^*$ and action-value function $Q \approx Q^*$.

```
/* Start of algorithm proper */
```

1 **while** *optimal policy not found* **do**
2   Generate an episode using π.
3   **foreach** *pair* (s, a) *appearing in the episode* **do**
4   r = return following the first-visit to (s, a)
5   Append r to R(s, a)
6     $Q(s, a)$ = average of all entries in array R(s, a)
7   **end**
8   **foreach** *state* s *in the episode* **do**
9     $a^* = \arg\max_a Q(s, a)$
10   **foreach** $a \in \mathcal{A}(s)$ **do**
11   **if** $a = a^*$ **then**
12         $\pi(a \mid s) = 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$
13   **else**
14         $\pi(a \mid s) = \frac{\epsilon}{|\mathcal{A}(s)|}$
15   **end**
16   **end**
17   **end**
18 **end**

visited. This assumes an every-visit method; for first-visit methods, $\mathcal{T}(s)$ would just have one entry per episode, which is the first time that state was visited during the episode. Let $T(t)$ be the time that the episode in which time t falls eventually terminates, and R_t be the return after time t (i.e., until time $T(t)$). Then $\{R_t\}_{t\in\mathcal{T}(s)}$ is the set of all returns after visits to state s and $\{\rho_t^{T(t)}\}_{t\in\mathcal{T}(s)}$ the set of corresponding importance-sampling ratios. Then to form an estimate $V(s) \approx V_\pi(s)$ we use the importance-sampling ratios to scale the returns and average the results,

$$V(s) \coloneqq \frac{\sum_{t\in\mathcal{T}(s)} \rho_t^{T(t)} R_t}{|\mathcal{T}(s)|}.$$

This is ordinary importance sampling. We can also weight the average, which is known as *weighted importance sampling.* Here, we instead use

$$V(s) \coloneqq \frac{\sum_{t\in\mathcal{T}(s)} \rho_t^{T(t)} R_t}{\sum_{t\in\mathcal{T}(s)} \rho_t^{T(t)}},$$

with $V(s) \coloneqq 0$ if the denominator is zero.

There are important differences between the two kinds of importance sampling, which touch on another concept that recurs repeatedly throughout reinforcement learning. This is the *bias-variance trade-off.* The ordinary importance sampling estimate is unbiased but has unbounded variance. On the other hand, the weighted importance sampling estimate is biased but the variance converges to zero. Therefore, in practice, the weighted one has lower variance and is usually preferred.

Suppose we have a sequence of returns $R_0, R_1, \ldots, R_{n-1}$, all starting in the same state and each with a corresponding weight W_i (for example, $W_i = \rho_t^{T(t)}$). If we want to estimate

$$V_n \coloneqq \frac{\sum_{k=0}^{n-1} W_k R_k}{\sum_{k=0}^{n-1} W_k}$$

and update it when we observe the next return R_n, we must also record, for each state, the cumulative sum C_n of the weights given to the first n returns. Then the update rule for V_n is

$$V_{n+1} \coloneqq V_n + \frac{W_n}{C_n}\big[R_n - V_n\big]$$

and

$$C_{n+1} \coloneqq C_n + W_{n+1},$$

where $C_0 = 0$ (and V_0 is arbitrary). Using this, we can evaluate a policy incrementally, from episode to episode.

We can now give a complete off-policy Monte Carlo control algorithm for estimating π^* and Q^*, based on generalized policy iteration and weighted importance sampling; this is Algorithm 6.3 below, which is a modified version of the algorithm given in [128, p. 119]. The target policy $\pi \approx \pi^*$ is the greedy policy with respect to $Q \approx Q_\pi$. The behavior policy can be almost anything but we still theoretically require that an infinite number of returns are obtained for each state and action pair in order to guarantee convergence of the policy to the optimal, which can be assured by choosing μ to be ϵ-soft. As before, we have omitted the termination check from the `while` loop for the sake of clarity, although we do this in the usual way by comparing successive approximations. The policy π converges to the optimal policy at all visited states even though actions are selected according to a different policy.

Algorithm 6.3: Off-policy every-visit Monte Carlo control.

```
/* Inputs and outputs                                              */
```

Input : Discount factor γ. Action-value function $Q(s, a)$ and function $C(s, a)$, arbitrary for all states s and actions $a \in \mathcal{A}(s)$. Deterministic policy π that is greedy with respect to Q.

Output: Updated policy $\pi \approx \pi^*$ and action-value function $Q \approx Q^*$.

```
/* Start of algorithm proper                                       */
```

1 **while** *optimal policy not found* **do**
2   Generate an episode (of maximum length T) using any soft policy μ.
3   R = 0
4   W = 1
5   **for** $t = T - 1, T - 2, \ldots, 0$ **do**
6         $R = \gamma R + R_{t+1}$
7         $C(s_t, a_t) = C(s_t, a_t) + W$
8         $Q(s_t, a_t) = C(s_t, a_t) + \frac{W}{C(s_t, a_t)}\big(R - Q(s_t, a_t)\big)$
9         $\pi(s_t) = \arg\max_a Q(s_t, a)$, with ties broken in a consistent manner.
10   **if** $a_t \neq \pi(s_t)$ **then**
11   EXIT FOR LOOP
12   **end**
13        $W = \frac{W}{\mu(a_t|s_t)}$
14   **end**
15 **end**

Chapter 7

Temporal-difference learning

One comment we can make about the Monte Carlo methods described in the previous chapter is that they do not bootstrap. In this chapter we introduce a family of methods that, like Monte Carlo algorithms, can learn from sampled experience, but also bootstrap, as in classic dynamic programming. These are *temporal-difference* methods, and their importance to modern reinforcement learning is such that Sutton and Barto state that, “if we had to identify one idea as a central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning” [128, p. 127]. As in the previous few chapters, here we closely follow the structure of [128, Chapter 6] and use similar notation (including *TD* for *temporal-difference*).

7.1 TD prediction

Like Monte Carlo methods, temporal-difference methods use the basic generalized policy iteration framework, in which policy evaluation and policy improvement steps interact in some way in order to achieve optimality. The major difference between the two is how they handle the prediction problem. Recall that

$$\begin{aligned} V_\pi(s) &\coloneqq \mathbb{E}_\pi[R_t \mid s_t = s] &(7.1)\\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s\right] \\ &= \mathbb{E}_\pi\left[r_t + \sum_{k=1}^{\infty} \gamma^k r_{t+k} \;\middle|\; s_t = s\right] \\ &= \mathbb{E}_\pi\big[r_t + \gamma V_\pi(s_{t+1}) \mid s_t = s\big]. &(7.2) \end{aligned}$$

For every state s_t, Monte Carlo methods essentially use equation (7.1) as the target for their estimate of $V_\pi(s)$, which is made by using a sample return instead of the real expected return. On the other hand, TD methods use equation (7.2)

as their target; thus they bootstrap because they use the current estimate of the value function rather than the actual function itself to make their estimates.

Whereas Monte Carlo methods need to wait until they have observed the return at the end of an episode to make their updates, TD methods can do so after every single time step; in practice, this can often make a big difference and they may be preferred for this reason. The most straightforward TD method is called *TD(0)* and works by making the update

$$V(s_t) = V(s_t) + \alpha\big[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\big], \tag{7.3}$$

where γ is the discount factor for the problem and α is a constant step size parameter. It is sometimes beneficial to work with the action-value function, in which case the update takes the form

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big]. \tag{7.4}$$

Pseudocode for a complete (tabular) TD(0) method for estimating V_π for a given policy π is given in Algorithm 7.1; this is a very slightly modified version of a similar algorithm given in [128, p. 128]. An almost identical algorithm can be given for estimating Q_π which uses equation (7.4) rather than (7.3) to learn the value of the state-action pairs.

Algorithm 7.1: TD(0) for estimating V_π.

```
/* Inputs and outputs                                              */
Input  : Discount factor γ. Constant step size parameter α. Value
         function V(s), arbitrary for all states s. Policy π to be
         evaluated.
Output: Updated V ≈ V_π.
/* Start of algorithm proper                                       */
1  foreach episode do
2      Initialize current state s
3      foreach step of episode do
4          Take the action a = π(s) and observe reward r and next state s′
5          V(s) = V(s) + α[r + γV(s′) − V(s)]
6          s = s′
7          if s is terminal state then
8              EXIT FOR LOOP
9          end
10     end
11 end
```

The expression $r + \gamma V(s') - V(s)$ in the TD(0) update is known as the *TD error*, so-called because it gives the difference between the estimated value of

the state s and the improved estimate $r + \gamma V(s')$. Note that the TD error at time t cannot actually be calculated until time $t+1$, as it requires the next state and reward. It is possible to write the error in using the Monte Carlo estimate to approximate the value function as the sum of TD errors. This is important theoretically, but beyond our scope; more detail can be found in [128, Chapter 6].

It can be shown—although we will not do so here—that for any fixed policy π the TD algorithm is guaranteed to converge to V_π if the sequence of step size parameters α_n decreases such that

$$\sum_{n=1}^{\infty} \alpha_n = \infty \quad \text{and} \quad \sum_{n=1}^{\infty} \alpha_n^2 < \infty, \tag{7.5}$$

and that it converges in the mean for a sufficiently small constant α. The same convergence results hold for the action-value function variant. One thing that hasn't been proven is which of the TD or Monte Carlo methods converges more quickly, although in practice the former are often faster, at least for stochastic problems [128, p. 132].

7.2 Sarsa

The natural step now is to use TD prediction methods for policy evaluation in the generalized policy iteration framework. First, we consider an on-policy method. The basic idea, as always, is that we estimate Q_π for the current policy π and then improve π by making it greedy with respect to the latest estimate of Q_π. Recall from equation (7.4), that in order to learn the value of state-action pairs—and thus estimate Q_π—we need to make an update that uses the current state (s_t), the current action (a_t), the reward received (r_{t+1}), the next state (s_{t+1}) and the next action (a_{t+1}). Because of this structure, the algorithm is known as *State-Action-Reward-State-Action* or, more commonly, *Sarsa*. Algorithm 7.2 gives a pseudocode for the Sarsa algorithm, which is a modified version of that given in [128, p. 138]. The Sarsa algorithm converges provided that all state-action pairs are visited an infinite number of times and that the policy converges in the limit to the greedy policy (which can be guaranteed for e.g. ϵ-greedy policies by setting $\epsilon = 1/t$, where t is the current time) [128, p. 138].

7.3 Q-Learning

An off-policy control algorithm can be defined by replacing the update (7.4) used in the Sarsa algorithm with

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\big]. \tag{7.6}$$

Algorithm 7.2: Sarsa algorithm.

```
/* 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                                             */
```

1 **foreach** *episode* **do**
2   Initialize current state s.
3   Choose action a from s using some policy derived from Q (e.g., ϵ-greedy).
4   **foreach** *step of episode* **do**
5   Take action a and observe reward r and next state s'.
6         $Q(s, a) = Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$
7         $s = s'$; $a = a'$
8   **if** *s is terminal state* **then**
9   EXIT FOR LOOP
10   **end**
11   **end**
12 **end**

Initially proposed by Watkins in his PhD thesis in 1989 [150], this algorithm is known as *Q-learning* and its discovery was one of the seminal events in modern reinforcement learning. Watkins revisited the algorithm with Dayan three years later [149], when 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 that a similar result to (7.5) holds for the sequence of step sizes α_n.

Algorithm 7.3: Q-learning for off-policy control.

```
   /* Inputs and outputs                                              */
   Input  : Discount factor γ. Constant step size parameter α.
            Action-value function Q(s, a), arbitrary for all states
            s ≠ terminal state and actions a ∈ 𝒜(s), with
            Q(terminal state, ·) = 0.
   Output: Updated Q ≈ Q*.
   /* Start of algorithm proper                                       */
 1 foreach episode do
 2 |  Initialize current state s.
 3 |  Choose action a from s using some policy derived from Q (e.g.,
   |   ε-greedy).
 4 |  foreach step of episode do
 5 |  |  Take action a and observe reward r and next state s'.
 6 |  |  Q(s, a) = Q(s, a) + α[r + γ max_a' Q(s', a') − Q(s, a)]
 7 |  |  s = s'; a = a'
 8 |  |  if s is terminal state then
 9 |  |  |  EXIT FOR LOOP
10 |  |  end
11 |  end
12 end
```

7.3.1 Expected Sarsa

Through a slight modification to the Q-learning algorithm, we can construct an off-policy extension of Sarsa. Known as *expected Sarsa*, the algorithm is almost identical to Q-learning but instead uses the update rule

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big[r_{t+1} + \gamma\mathbb{E}[Q(s_{t+1}, a_{t+1}) \mid s_{t+1}] - Q(s_t, a_t)\big]. \tag{7.7}$$

Practically, this is more complex to implement computationally than the original Sarsa algorithm, but removing the random choice of the next action a_{t+1} reduces the variance, so in general expected Sarsa tends to perform slightly better than the original (and often Q-learning as well) [128, p. 143].

7.4 Double learning

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

We can see immediately from Algorithm 7.3 that Q-learning requires a maximization and so may be susceptible to maximization bias. It is less obvious that Sarsa is affected but in fact it generally is as well since it often uses ϵ-greedy policies which also require a maximization. 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. We can easily incorporate double learning into any of the TD methods we have introduced already; a double Q-learning method is described by Algorithm 7.4 below, which is a slightly modified version of that given in [128, p. 144].

Algorithm 7.4: 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 */
```

1 **foreach** *episode* **do**
2 Initialize current state s.
3 Choose action a from s using some policy derived from Q_1 and Q_2 (e.g., ϵ-greedy in $Q_1 + Q_2$).
4 **foreach** *step of episode* **do**
5 Take action a and observe reward r and next state s'.
6 $r =$ random number from $(0, 1)$
7 **if** $r < 0.5$ **then**
8 $Q_1(s,a) = Q_1(s,a) + \alpha\big[r + \gamma Q_2\big(s', \arg\max_{a'} Q_1(s',a')\big) - Q_1(s,a)\big]$
9 **else**
10 $Q_2(s,a) = Q_2(s,a) + \alpha\big[r + \gamma Q_1\big(s', \arg\max_{a'} Q_2(s',a')\big) - Q_2(s,a)\big]$
11 **end**
12 $s = s'$
13 **if** *s is terminal state* **then**
14 EXIT FOR LOOP
15 **end**
16 **end**
17 **end**

Chapter 8

Function approximation and deep reinforcement learning

When we consider the task scheduling problem in high-performance computing, it soon becomes clear that the number of possible states is likely to be very large, perhaps to the extent that it is impossible to actually store the value of every possible state and action pair. 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 the state space is likely to be very large (our approach toward expressing the problem as an MDP is something will be discussed in more detail in Section 9.4.1).

In order to define the state at any one time in a useful way, we ideally need to include which tasks have already been executed, which are currently being executed (and by which processor), which are already scheduled (and which processor they are scheduled on) and which haven't been scheduled, among potentially much else. 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 very large; for example, if our actions 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 dependencies) on the order of the number of tasks multiplied by the number of processors, both of which are likely to be large.

Almost all interesting real world problems also must contend with this problem of impractically large state spaces, which is ultimately of course just the curse of dimensionality from Chapter 5 once again. In such problems, we have no chance of finding truly optimal policies and value functions, but must make do with good approximations; hence methods for handling them are often known as *approximate dynamic programming*. The key idea behind this approach is *function approximation*: since there are far too many possible state and action pairs for us to be able to explicitly find $Q(s, a)$ for all of them, we instead extrapolate from a feasible number of samples in order to estimate function values for

all pairs. The obvious problem here is how we can possibly say anything about states and actions we haven't encountered before, so the trick is being able to generalize what we do know as far as possible.

There are basically two different ways approximate value functions can be obtained:

1. Solve a smaller, more tractable (but related) problem and use the value function computed there as an approximation of the value function for the original, larger problem.

2. Find a suitable parameterized function $\tilde{Q}(s, a, r)$ to approximate $Q^*(s, a)$, for all states s and actions $a \in \mathcal{A}(s)$, where r is a vector of tunable parameters (or *weights*), usually obtained through optimization.

The latter approach is the more theoretically robust so it is the one that we will consider exclusively from now on, in which case there are two further choices to be made:

1. What kind of approximation architecture to use for representing the value function.

2. How we update the parameter vector r.

A popular choice of approximation architecture is *artificial neural networks* (which are the focus of the next section) so methods which make use of them are often called *neuro-dynamic programming* [16], although this is another area in which the terminology varies depending on the source. In Section 8.3 we describe some notable recent successes which were achieved through use of neural networks—and in particular *deep* neural networks—for this purpose, and which suggest that, at least for certain problems, the curse of dimensionality can be overcome. First, however, in the next two sections of this chapter we give a brief introduction to neural networks and how they may be used for function approximation in reinforcement learning.

8.1 Neural networks and deep learning

Artificial neural networks are widely used for approximating nonlinear functions [128, 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*. For reasons that we will not discuss here, these activation functions are often sigmoidal. Popular choices therefore are the logistic function,

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

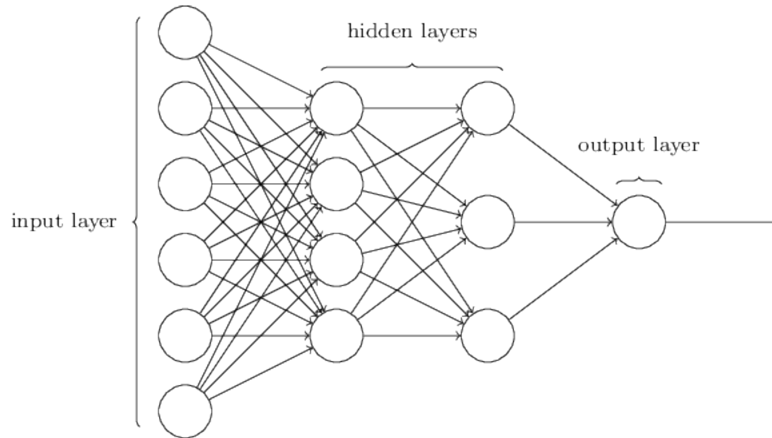

Figure 8.1: A generic feedforward neural network. Source: [85].

or the arctangent function

$$f(x) = \tan^{-1}(x),$$

or, more simply,

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

Neurons which make use of the last activation function are known as *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. A typical example of a feedforward neural network is shown in Figure 8.1. 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 [64].

Often it useful for the output of the final layer of the network (and sometimes other layers) to represent probabilities; in other words, we want the output of each neuron in the layer to be between zero and one, and we want the sum of the outputs of all the neurons in the layer to be one. In this case the *softmax* activation function is often used. Suppose the ith neuron in the ℓth layer is denoted by a_i^ℓ, and that the weighted sum of the input of neuron a_i^ℓ is given by

$$z_i^\ell = \sum_k w_{ik}^\ell a_k^{\ell-1} + b_i^\ell,$$

where w_{ik}^{ℓ} is the weight from neuron i in layer ℓ and neuron k in layer $\ell - 1$, and b_i^{ℓ} is a quantity known as the *bias*[1] associated with neuron i in layer ℓ. Then using the softmax activation function, the output of the neuron a_i^{ℓ} is given by

$$a_i^{\ell} = \frac{e^{z_i^{\ell}}}{\sum_k e^{z_k^{\ell}}}.$$

Another important concept with regards to a deep neural network 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* [85, 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*; a brief guide to gradient descent and its variants, with a particular emphasis on machine learning applications, can be found at [104].

Regarded as a major breakthrough that revitalized the field when it was initially discovered, backpropagation may not actually work that well for deeper neural networks [13] and alternatives like *dropout*—in which we randomly remove neurons from the network in order to reduce *overfitting* of the training data [118]—have also become popular in recent years [128, Chapter 9].

Deep neural networks are often referred to as "universal approximators" [128, 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 [29].

Machine learning methods which make use of deep neural networks are part of a field called deep learning[2]. Like reinforcement learning, deep learning is currently undergoing something of a renaissance and has been applied to a wide range of problems, with many noteworthy achievements in areas like image recognition. A thorough survey of the current trends in deep learning can be found at [112], and a definite recent guide to the field as a whole at [45].

8.2 Deep reinforcement learning

Any reinforcement learning method in which deep neural networks are used is known as *deep reinforcement learning* (i.e., it is almost precisely the same as

[1] This is important in general, but not for us, so we won't discuss it any further.

[2] Technically, deep learning is not exclusively tied to artificial neural networks, but we will not consider any of the alternatives.

neuro-dynamic programming). The general approach is exactly that described at the start of this chapter: we want to approximate the value or action-value function of a problem and use that in a reinforcement learning algorithm. 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, for the most part, is *yes*. Many of the reinforcement learning algorithms we have introduced so far have been successfully extended to large-scale problems through the use of deep neural networks to approximate their value functions; we will see examples in the next section. This is an area in which the theory tends to lag far behind practice, but formal convergence results and performance guarantees have also been proven. The most mathematically rigorous treatment of the subject is by Bertsekas and Tsitsiklis [16], which (arguably) lays down the theoretical bedrock on which the rest of the subject is built. We are not going to discuss the mathematical basis of deep reinforcement learning in any greater depth here, but it is something we have studied and will need to be conscious of when we attempt to apply deep reinforcement learning algorithms to the task scheduling problem in HPC, as is our intention (see Chapter 9).

8.3 Notable successes

Historically, reinforcement learning was generally hindered by many of the issues that we have already mentioned—the curse of dimensionality, achieving sufficient exploration, long-term credit assignment, and so on—and unable to achieve any significant success with truly interesting real world problems until methods that incorporated neural networks for function approximation became popular. A partial aversion of this was perhaps Arthur Samuel's famous checkers-playing program from the late 1950s [107] (improved continually in the years following [108]), which presaged some ideas from modern reinforcement learning (and temporal-difference learning in particular), although in a different framework. Samuel's program was eventually able to achieve a level of performance at Checkers similar to a decent amateur human player, which was remarkable for the time [128, p. 430].

The first game-playing reinforcement learning agent to achieve a level of play beyond that of all but the strongest human players was Tesauro's TD-Gammon program for backgammon [130, 131]. Not coincidentally, this was also the first prominent effort to use deep neural networks for value function approximation, in this case a relatively simple network with just one hidden layer. The reinforcement learning algorithm used was a modified version of a standard TD learning algorithm called *TD(λ)*, which we have not heretofore discussed but which can, roughly, be regarded as a generalization of the TD(0) algorithm from Section 7.1 that makes use of *eligibility traces*—essentially, a temporary record of previous events—in an attempt to deal with the credit assignment problem [129].

For training, TD-Gammon played games against itself, a technique which has often been imitated since, achieving a level of play roughly equal to the most successful backgammon programs then in existence—all of which had some human expert input—after about 300,000 such games. Tournament matches against human grandmasters throughout the 1990s confirmed that TD-Gammon was playing at approximately their level, and its success had such an effect that human players began to imitate its approach toward aspects of the game [132]. Similarly, because of the relatively complex nature of backgammon—which has approximately 10^{20} possible board configurations—TD-Gammon showed the potential of deep reinforcement learning and inspired much future research in the area.

The reader will note that both of the examples mentioned so far have concerned games. Indeed, this has proven to be the area in which deep reinforcement learning algorithms have achieved their greatest successes, and in the next two sections we describe two more recent efforts in this area, both of which have received widespread attention and have had an invigorating effect on the field. However, a word of caution is perhaps in order at this point with regards to our stated aim of applying (deep) reinforcement learning to the task scheduling problem in HPC: real-world applications of deep reinforcement learning remain rare, and it is very much still an emerging subject. This will be revisited in the next chapter.

8.3.1 Mastering Atari games

Arguably the most prominent achievement in all of machine learning in recent years was the success researchers at Google DeepMind [74] had in training a deep reinforcement learning agent to play a collection of video games for the Atari 2600 console at superhuman levels, which attracted notice far beyond academia [81]. Published in Nature in 2015 [79], their work suggested that, at least in certain cases, it was now possible for modern reinforcement learning agents to both learn how to perform well on complex high-dimensional problems and transfer their learning across a wide variety of different tasks.

The DeepMind team created a novel reinforcement learning agent they called a *deep Q-network (DQN)*, which integrates Q-learning with a deep convolutional neural network for value function approximation. The choice of network architecture was natural because of the fact that the DeepMind team chose to present game data to the agent as video (i.e., images in quick succession), equivalent to 50 million frames for each game; this itself was a major achievement—that the agent was able to learn each of the games from raw images, without any game-specific adjustments.

DQN differed in several ways from standard Q-learning. One of the most important was *experience replay*, a technique first studied by Lin [70], which was inspired by theories of how the hippocampus aids learning in the human brain. The basic idea is that the agent's experience (current state, chosen action, reward,

next state) at every time step of every episode are pooled into a large data set, and that the action-value function updates in the Q-learning algorithm are made based on samples drawn (uniformly) at random from this set. Experience replay has often been found, as it was here, to improve learning by preventing overfitting and avoiding becoming trapped in local optima.

Tested on a suite of 46 Atari 2600 games, the DQN agent was able to meet or exceed human performance (defined as achieving more than 75% of the score of a human tester who had practiced on the game for two hours beforehand) on 29 of them, sometimes by a wide margin, despite the fact that some of them were very different. This generalization was perhaps the most novel and remarkable aspect of the DeepMind team's work.

8.3.2 AlphaGo

Go is one of the oldest board games known but until recently had successfully resisted all attempts to create artificial agents capable of human levels of play, in part due to the truly enormous number of possible different board configurations (approximately 10^{170} [140]). However, in 2016, a team of researchers at Google DeepMind (with some personnel in common with the team that designed the Atari playing agent) created *AlphaGo*, a learning program that was soon able to not just achieve human levels of play but emphatically defeat the reigning European Go champion [115].

Unlike the Atari controller, AlphaGo made use of a large database of moves by human experts so incorporated *supervised*, as well as reinforcement, learning, in addition to deep neural networks. However, a year later, in 2017, DeepMind released AlphaGo Zero [116], which was capable of learning based only on experience without any human input and was therefore a wholly reinforcement learning based agent. One hundred head-to-head games suggested that AlphaGo Zero not only matched AlphaGo but actually exceeded its performance, winning every single one.

We will not actually describe how AlphaGo (and AlphaGo Zero) work here, but we mention them to emphasize that state-of-the-art reinforcement learning methods are capable of handling problems with enormous state spaces.

Chapter 9

Outline of our approach and progress so far

We believe that applying reinforcement learning to the task scheduling problem in high-performance computing is a promising avenue that should be investigated; this is the major aim of this PhD research. How we intend to pursue this investigation and the progress we have made thus far are the focus of this chapter.

Using machine learning—and reinforcement learning in particular—to tackle various scheduling problems has become increasingly popular in recent years; we chronicle many such efforts in Section 9.5. Some of these applications have similarities with the task scheduling problem which suggest that it may also be tractable by those methods. Until recently the major issue with applying reinforcement learning to a problem such as ours was the curse of dimensionality, the fact that our state space is likely to be impractically large. However, the recent progress made by combining reinforcement learning methods with deep neural networks chronicled in the previous chapter suggest that it is possible that this may be overcome.

9.1 What do we really want?

As with any project, it is important to first clearly establish what we actually hope to achieve. Basically, we want to use reinforcement learning to find a good quality schedule for any application in any computing environment. But the question is, what does a solution to this problem look like?

Suppose for instance that we want to find the optimal way to schedule an application which performs the Cholesky factorization of a matrix on a given HPC system. In that case what we want from a reinforcement learning algorithm is that it takes the application (or its task graph) as input and returns a good schedule after an acceptably small amount of training. A good schedule can be defined however we choose it to be; the most natural choice would be that the

best schedule is the one which minimizes the makespan, the time it takes to execute all the tasks in the DAG. But we may want to minimize, for example, energy consumption instead, or perhaps minimize the makespan whilst also keeping energy costs below a prescribed limit. Ideally, we should like a reinforcement agent that can take the variable we want to optimize as an input and return a schedule that optimizes that variable. Achieving such a level of generalization however will surely be tricky and is not something we have begun to seriously consider yet.

Then we want to find the best such reinforcement learning algorithm, that returns the best quality schedule with the least amount of training, or that balances the two in the most acceptable way. Hence one of the aims of this work is to evaluate the performance of existing reinforcement learning algorithms, such as those described in Chapters 6, 7 and 8, for this purpose, comparing them both to each other and alternative approaches such as HEFT for a wide range of different applications and a variety of different system architectures.

Our intention then is to use the observations we make and the information we glean from experiments with existing reinforcement learning algorithms in order to design novel ones for the task scheduling problem. This will involve identifying the key features of the algorithms that are most important for ensuring good performance, as well as theoretical analysis of the characteristics of the problem itself. Our hope then is that our handcrafted algorithms will be able to improve upon the performance of both existing scheduling heuristics and reinforcement learning algorithms.

We need to always remain practical. Even if we have a reinforcement learning algorithm that can learn a good schedule for a given application on a given machine, what we ideally want to be able to do is find some way to transfer what we have learned from one application and one machine to other applications, run on different machines; this, after all, is what learning really means. Another goal of our research then is to develop ways to transfer this knowledge. The (perhaps impossible) ideal here is a scheduling algorithm that, after a short period of training on a diverse set of test applications, is able to almost immediately achieve good schedules for any application run in the computing environment in the future.

9.2 An immediate problem

Perhaps the major practical concern that leaps out when we consider applying reinforcement learning to the task scheduling problem is that reinforcement learning methods usually need to do lots of training but on HPC systems this may be prohibitively expensive; after all, the Google DeepMind Atari controller required the equivalent of around 38 days of training [79], a small amount perhaps compared to the experience accrued by human experts over their lifetime but a long

time in a HPC context, where the overriding concern is to minimize computing time in order to reduce overall operating costs. It is no coincidence that reinforcement learning has achieved its greatest success with games, where training data is relatively cheap and easy to acquire. We also have to seriously consider whether the cost of the training negates any gains we may achieve by obtaining a more optimal schedule; we saw in Section 3.2.2 that one of the drawbacks of using genetic algorithms for scheduling problems is that, although the quality of the schedule achieved is often much better than for competing methods, the time it takes to obtain it can be prohibitive.

Reducing the amount of training required to achieve good results is obviously one of the major goals for reinforcement learning methods in general. We really want *data efficient* algorithms that can extract as much useful information from the data already gathered as possible. An interesting application area in which data efficiency is particularly important is robotics, where for practical reasons it may be difficult to gather the data necessary for traditional reinforcement learning techniques. A policy search reinforcement learning method known as *Probabilistic Inference for Learning COntrol* (PILCO) [33] has recently made some promising progress toward data efficient learning in this area [34], so we briefly considered if it may also be suitable for our purposes.

PILCO is a model-based method and typically the biggest problem with that approach is overconfidence in the accuracy of the model. To overcome this, PILCO (and other probabilistic model methods) take into consideration the level of confidence we have about the model itself. PILCO uses nonparametric Gaussian processes as a probabilistic model and treats uncertainty in the model as noise. With this approach, PILCO does not require an explicit value function, unlike most of the other methods we have introduced so far. PILCO is currently limited to episodic systems, but extension to the more general case may well be possible.

Although the focus on extracting as much information from the data we have as possible is certainly appealing, one thing PILCO and similar algorithms don't handle particularly well is large state spaces such as in our task scheduling problem, so we have no plans to investigate PILCO or similar probabilistic model methods in greater depth at the moment. However, it is something we may wish to revisit at some point in the future.

9.3 Our solution: simulation

As already noted, the standard approach to gathering training data when real data is difficult to acquire is simulation. The use of simulation for reinforcement learning applications is well-established and has a solid theoretical foundation [16]. From a practical standpoint, a simulator offers us several potential advantages over training on real HPC systems. Depending on the capabilities of our

simulator, we may be able to consider a wide variety of potential architectures and applications. Observing how an algorithm performs on a totally different architecture thus becomes as easy as adjusting the parameters of the simulator, rather than trying to secure access on a real machine. We also do not have to worry about the operating costs associated with running real HPC machines; we can do as much training as we desire.

Of course, the obvious issue with using a simulator is that the data we obtain may not be truly reliable. Fortunately, good software for simulating the behavior of arbitrary computing environments exists. One such option that we have explored is the focus of the next section.

9.3.1 SimGrid

Since we have previously made use of StarPU (see Section 2.4.2), the natural choice of simulator is SimGrid [23], which has been integrated with StarPU in the last few years; collaboration between the StarPU and SimGrid teams is ongoing and is intended to be pursued further [119, 120, 121]. SimGrid is a fast, mature, open-source, `C/C++` language software simulator that allows the reliable simulation of application execution on user-defined parallel, distributed computing platforms [3, 18, 40, 60, 66, 100, 101, 126, 143].

The advantage of the integration of StarPU and SimGrid is that we can write efficient application code with StarPU and then use SimGrid to simulate its execution on a wide variety of different architectures. SimGrid allows users to specify an almost arbitrary range of target architectures, with users able to add as many processors as they like, of whatever speed they choose, and describe how they are linked. The typical way to do this is in a separate `XML` file. From a practical standpoint, we should note that it is somewhat awkward to construct the extremely complex systems typical of modern HPC in `XML`. The SimGrid development team is aware of this and also permits the use of `Lua` files to describe environments instead; the intention is that eventually users will be able to describe their chosen environment entirely in `C/C++` [117]. In the future, we also intend to investigate if we can find some way to more easily define computing environments in SimGrid. As a final point, we note that the ability to define arbitrary architectures is useful not just for training a reinforcement learning agent for an existing HPC system, but also offers the potential of identifying key features that we should include when designing future system architectures.

Ultimately our intention is that we write our application code in StarPU and that the runtime then constructs the DAG. However, SimGrid also includes the SimDag interface, an environment for dealing with applications represented as DAGs, which allows us to simply deal with the task graph of an application without having to consider the application itself. There are both advantages and disadvantages to this approach: we can generate random DAGs easily, certainly more so than writing new code for an application from scratch, but we need to

ensure that they are representative of the kind of applications we want to consider. SimDag allows us to construct arbitrary DAGs, either directly in our `C/C++` code or by loading them from a `DOT` or `DAX` file; we have generally found the latter to be easier. We can specify the type (e.g., communication or computation) of all the tasks in the DAG, as well as their size, and all the dependencies between them. Generating random DAGs isn't difficult, even from scratch, although good open-source software exists, such as `DAGGEN`, created by Suter (a member of the SimGrid development team), the source code for which is available at his Github page [125].

Both StarPU and SimGrid allow users to design their own scheduling strategies, giving us the opportunity to implement existing reinforcement learning algorithms and also any new ones we may develop. Thus far, we have mostly worked in SimDag towards this end, but our intention is to do the same in StarPU in the future. We have begun to implement some of the simple Monte Carlo and temporal-difference reinforcement learning algorithms described in Chapters 6 and 7, although our progress here has been slower than we would have liked. We have had some success very recently however and are hopeful that we will be able to successfully implement (tabular) versions of algorithms like off-policy Monte Carlo control, Sarsa and Q-learning in the SimDag framework in the immediate future. We should then be able to evaluate their performance on a variety of randomly generated DAGs and computing environments. After this, we will want to begin to move towards approximate methods such as those described in Chapter 8. From a practical standpoint, this will require surmounting some more implementation hurdles but we are optimistic that we will be able to meet them.

9.4 Some current and potential issues

In this section we detail some of the problems we have encountered thus far, and others that we are aware will need to be satisfactorily solved in the future but that we have not begun to seriously grapple with yet.

9.4.1 Characterizing the MDP

Implicit in the assumption that we can apply reinforcement learning to the task scheduling problem is that we can characterize it as a Markov decision process, with states, actions and rewards defined in a useful way that allow us to actually solve the problem we want to solve. But there are a lot of decisions we need to make in order to ensure that this is the case.

Roughly speaking, in any planning problem the state of the system should tell us all the information we need in order to make a decision. So it seems reasonable to suppose that the system in our problem is fully described at any point in time by the state of the processors and the state of the application (as we have mostly

been working practically in SimDag, we will use *DAG* from now on rather than *application*). The state of the DAG is defined by the state of all the tasks in it: whether they have already been executed, or have been scheduled, or are waiting to be scheduled, and so on. Similarly, the state of the processors could be a list of which tasks are currently scheduled on each of the processors and their costs. To be truly useful, we can see that the state space for our problem must be very large, at least in the case of DAGs with many nodes and HPC systems which have potentially millions of processors. So far, we have only experimented with small DAGs and simulated computing environments with a small number of processors, although our intention is to move on to larger, more realistic examples soon.

How we choose to define the actions we can take is not immediately apparent. Thus far in our experimentation we have typically chosen our actions to be scheduling a specific task on a specific processor, which usually seems reasonable, but there is plenty of scope for further investigation here. For example, in an environment comprising CPUs and GPUs it may be sensible to define our action to be which type of processor to schedule a task on (with some prescribed alternative action to take if all of the processors of the chosen kind are busy).

Similarly, choosing a useful reward function that encourages the agent to do precisely what we want it do can be notoriously tricky [54]. A recent example of this comes from a paper by Popov et al. [98]. There they were attempting to design a reinforcement learning controller for a robot arm to master the task of grasping a red block and stacking it on top of a blue one. Once the red block was successfully grasped, a reward was given for the controller's success in guiding it towards the top of the blue block. However, this reward was defined by the height achieved by the bottom face of the red block, so in some cases the controller learned to simply flip the block over!

If we wish to find, for example, the schedule that reduces the overall execution time, then our reward function must reflect this. Assuming our actions are defined by scheduling a specific task on a certain processor, then the most straightforward choice would be for the reward to simply be the (estimated) time it takes to execute the task on that processor, as in the HEFT algorithm. And of course, if we want to optimize something other than total execution time, such as energy consumption, then our rewards would have to be defined entirely differently (for example, estimated power consumption of that task on that processor). So far in our simple experiments we have focused on minimizing the overall makespan and have used simple rewards like estimated execution time, although we have also considered rewards defined as weighted sums of multiple variables that we want to optimize, such as estimated execution time and the number of new tasks which can be scheduled once the chosen task has been executed. This introduces the new problem of determining weights for the sum, which complicates matters further, and is probably not something we will pursue any further.

9.4.2 Transfer of learning

As we have already discussed, one of the major issues with applying reinforcement learning in general is *transfer of learning*—how do we transfer things we have learned from solving one problem to one we haven't seen before? For our problem in particular, we ideally want to be able to use the information we have gained from learning how to efficiently schedule a specific application on a specific platform to other applications, potentially on different platforms as well.

For example, testing on randomly generated DAGs was the approach taken by the creators of HEFT [138] to evaluate its performance and is a useful metric to some extent. But of course there is no substitute for the real thing, so there's no guarantee that an algorithm that performs well on a variety of random DAGs will perform just as well on real application task graphs. Perhaps the compromise solution then is to find some way to ensure that the DAGs we generate for training are not entirely random but are restricted in some sense to be useful for the applications we are studying—for example, if we want to optimize task scheduling for the Cholesky factorization of a matrix, we need to find some way to generate random DAGs which are similar to Cholesky factorization task graphs. If the DAGs we train on are sufficiently representative, we can potentially do all of our training this way. Precisely how to generate random instances of DAGs which represent specific applications is an open question that we intend to study further.

Similarly, of course, we would like to be able to apply things we have learned from learning an application on a specific system architecture to other architectures, but this is again not straightforward. One idea we have considered is doing some sort of clustering to divide all the architectures we might encounter into a small number of different categories, then training on a typical example of each category to find the best scheduling policy for the application on each category of architecture. This still leaves us with the problem of transferring learning between architectures in each category, but the idea is that this will be an easier problem to solve than the original one. This also is something we want to investigate further in future.

9.4.3 Remaining practical

This is another thing that we have already touched on but feel we should emphasize once again.

We are trying to solve a practical problem, so we always need to keep the big picture in mind and make sure that our solution is actually worthwhile. In particular, as we have already alluded to, we need to ensure that any gains we achieve from obtaining better schedules are worth the additional overhead incurred compared to the schedules we may obtain by other means. Of course, we are also trying to minimize this overhead, but it is likely our reinforcement learning approach is always going to need more computing time to obtain a

schedule than, say, HEFT. The question that must be asked is, are the potential gains worth the extra time and effort?

This research began with a focus on dense linear algebra, where existing scheduling heuristics generally perform quite well, and finding slightly better schedules may not realistically be worth the extra effort required to find them (although this just a possibility, and we still fully intend to investigate this). However, our approach may ultimately be better suited to an area like sparse linear algebra where making efficient use of the available resources is trickier and the quality of the schedule we obtain can make a bigger difference. Considering other potential applications for which finding good quality schedules is particularly important is another thing that we intend to do in the future.

9.4.4 Designing new algorithms

As previously stated, perhaps our ultimate aim with this project is to design novel reinforcement learning agents that can improve upon existing task scheduling techniques. Thus far, we have not begun to seriously consider how we will do this in detail. The original intention was to create a computational framework in which to implement many of the existing scheduling heuristics (e.g., HEFT) and the reinforcement learning methods that we wished to consider, and that we would then perform numerical experimentation in order to guide our analysis of the task scheduling problem and therefore help us to both adjust existing algorithms and design new ones. This is still the plan, in part, but setting up such a framework has proven to take longer than we had anticipated.

Our intention now then is to continue to try and get the computational framework up and running, but also to immediately begin to analyze the problem in a more mathematically rigorous way than we have thus far. One particular avenue we want to pursue in the immediate future is to consider the HEFT algorithm in detail and investigate how we can incorporate reinforcement learning to improve the estimation of task completion times.

9.5 Related work

The idea of applying machine learning to the problem of scheduling tasks on computational resources has been considered before, albeit perhaps not widely. The use of reinforcement learning—in particular, Q-learning—for dynamic load balancing in distributed heterogeneous systems was investigated by Parent et al. [94], and later Samreen and Kumar [106]. Li et al. [67] used artificial neural networks to estimate task execution times in their extension of the existing (static and restricted to homogeneous systems) *Modified Critical Path* (MCP) scheduling heuristic from [155]. The idea of training neural networks to predict thread performance on the diverse processing cores of a heterogeneous system in or-

der to maximize total throughput is also considered in the more recent work of Nemirovsky et al. [83]. Interestingly, they found that using deep neural networks sometimes led to diminishing returns compared to using more lightweight networks.

Ipek et al. propose a reinforcement learning based memory controller for multicore processors [53]. Particular attention is given in their paper 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 the controller was capable of out-performing the existing approaches at that time, it 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 et al. [102]. A Q-learning agent is proposed for routing jobs in ad-hoc networks in [25]. Tesauro investigates the use of reinforcement leaning agents for managing computer resource allocation in data centers in [133], as do Vengerov and Iakovlev in [144]. Negi and Kumar considered the use of machine learning to optimize the CPU time allocated to programs in Linux systems [82]. Galstyan, Czajkowski, and Lerman studied the application of reinforcement learning algorithms for resource allocation in grid computing environments [41]. Machine learning approaches have also been investigated for compiler optimization [124, 139].

In addition there is an extensive body of literature concerning the use of machine learning for general scheduling problems [9, 14, 22, 80, 99, 111, 113], and the job shop scheduling problem in particular [8, 49, 65, 103, 162], of which our task scheduling problem can be considered a specific example, albeit with important idiosyncrasies (see Section 2.5).

9.6 Summary of progress so far

In this section, I briefly summarize what I have actually accomplished over the first year of this PhD project, with more of a personal perspective than before.

- It was necessary to do a lot of study in order to become familiar with all the relevant topics, particularly as I don't have a strong background in, for example, applied probability. I have studied several courses, both for credit and online, to this end; a list is provided in Appendix A. Similarly, I have done a lot of reading and independent study, for which the references for this document are probably the best guide to what this comprised. These studies took up a good deal of my time this year, although will hopefully stand me in good stead over the next few years of this research.
- I made some progress toward setting up a computational framework which

will allow me to consider the application of reinforcement learning to the task scheduling problem. However, progress here has not been as rapid as I would have liked and there is still work to be done, with no real useful results produced so far. Very recently, I have had some success here and I am optimistic that I will soon be able to implement many of the tabular reinforcement learning algorithms that I want to apply to the task scheduling problem. More work will need to be done in the future to make the move toward implementing deep reinforcement learning methods for larger-scale problems, but I don't foresee any major issues there at the moment.

- So far I have not begun to seriously consider how existing task scheduling heuristics and algorithms can be extended with the use of reinforcement learning, nor the design of novel ones of my own. This is perhaps the area where I am most eager to make some progress, and I intend to do a lot of work here in the coming months (see Section 9.7).

9.7 Proposed timetable for future research

In this section I give a provisional timetable for the work I intend to do over the next year or so. This is of course subject to change, but should hopefully give a good idea of my overall plans for the remainder of the project.

Summer/Autumn 2018

- Successfully implement existing reinforcement learning algorithms like off-policy Monte Carlo control, Sarsa and Q-learning in SimDag, and analyze the quality of the schedules obtained for randomly generated DAGs and typical HPC environments.
- Begin to analyze the task scheduling problem in a more rigorous manner. Investigate how the HEFT algorithm may be improved by using reinforcement learning techniques. Use numerical experiments to guide the analysis as far as possible.

Autumn/Winter 2018

- Extend simulation framework so we can begin to implement and analyze existing deep reinforcement learning algorithms.
- Design and implement novel reinforcement learning agents for the task scheduling problem, assuming for the moment that we already have the DAG of an application. Analyze the results and use this to aid future designs.

2019 and beyond

- Consider the problem of how we can transfer learning from one DAG to another, or from one environment to another. How far can we take this? Investigate similar existing techniques and see if/how they may be adapted.

- Continue development and refinement of, and numerical experimentation with, prototype scheduling algorithms. Move beyond the assumption that we have been given the DAG and consider the complete process, from the application code development stage (i.e., integrate more fully with StarPU). This will be necessary in order to run application code on real HPC systems, which will allow us to verify the results gathered using SimGrid and demonstrate the real world potential of our algorithms.

Appendix A

Personal development

Courses taken

MATH69122 Stochastic Control with Applications to Finance

Internal course.
Course instructor: Neil Walton.

Although intended for MSc students studying mathematical finance and with most examples and applications discussed being of that nature, the bones of the course still provided a good general introduction to dynamic programming and Markov decision processes, with other relevant topics such as reinforcement learning also touched upon.

MAGIC099 Numerical Methods in Python

MAGIC course, Autumn 2017/18, 10 hours.
Course instructor: Ian Hawke, University of Southampton.
Grade: Pass (pass/fail marking scheme).

Reinforced my understanding of the basics of numerical analysis, and practical exercises allowed me to gain more experience with Python, which has become the most popular programming language for machine learning in recent years.

Heuristics and Approximation Algorithms

National Taught Courses in Operations Research (NATCOR) course.
University of Nottingham, 9th–13th April, 2018.

Short lecture course given by several experts in their respective fields. A general introduction to how heuristics are used to find good solutions to otherwise

intractable optimization problems. Specific topics covered in depth include metaheuristics such as simulated annealing, tabu search and genetic algorithms.

Convex Optimization

National Taught Courses in Operations Research (NATCOR) course.
University of Edinburgh, 4th–8th June, 2018.

A week-long lecture course given by prominent researchers in the field. After a brief introduction to the underlying principles of convex optimization, the focus then shifted to extended discussion of more specific topics, such as interior point methods and convex relaxation. Computational practicalities were emphasized, such as the importance of taking advantage of sparsity and matrix structure in order to efficiently perform the linear algebra operations required in, for example, both the simplex and interior point methods.

Online courses studied

In addition to the courses given above, I also studied the following online courses. All of the courses listed below consisted of online lecture videos and other resources such as written notes and assignments.

Machine Learning

Given at the University of Oxford, 2014/15.
Course instructor: Nando de Freitas.

A general but comprehensive introduction to machine learning, with some discussion of reinforcement learning in particular, by one of the most prominent experts in the field, who now also works for Google's DeepMind and has been involved in some of their recent achievements which we have already noted.

Reinforcement Learning

Given at University College, London, 2015.
Course instructor: David Silver.

Introductory lecture course in reinforcement learning, given (like the above) by a member of the Google DeepMind team who is prominent in the field and worked on some of their notable recent achievements. Covered all of the topics described in Chapters 6, 7 and 8 in greater detail than we gave there, and extended them considerably further.

Approximate Dynamic Programming

Given at Tsinghua University, 2014.
Course instructor: Dimitri Bertsekas.

Short six-lecture, twelve-hour course giving an overview of approximate dynamic programming for infinite-horizon problems.

Other activities

Society for Industrial and Applied Mathematics (SIAM)

Treasurer, University of Manchester Student Chapter.
October 2017–Present.

Responsible for receiving and dispersing all Chapter funds, and submitting periodic financial reports accounting for them to the parent organization.

Appendix B

Numerical linear algebra primer

Algorithms

Algorithm B.1: A practical algorithm for computing the block Cholesky factorization of a matrix using LAPACK functions.

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 $A = \begin{bmatrix} L_{11} & \\ L_{12} & L_{22} \end{bmatrix}$.

1 **while** A *is not fully factorized* **do**
2 Update $A_{11} := L_{11} = \texttt{POTRF}(A_{11})$.
3 Compute $A_{12} := L_{12} = A_{12} L_{11}^{-T}$ using `TRSM`.
4 Compute $A_{22} := L_{22} = A_{22} - L_{21} L_{21}^T$ using `SYRK` and `GEMM`.
5 $A := A_{22}$.
6 **end**

Here we have slightly simplified matters by assuming that the matrix is only partitioned into four blocks but the principle is the same if we partition the matrix further.

Software

BLAS (*Basic Linear Algebra Subprograms*) is a specification for extremely efficient, portable routines that perform basic linear algebra operations. Level 1 BLAS perform scalar, vector and vector-vector operations; Level 2, matrix-vector operations; and Level 3, matrix-matrix operations. They are the standard building blocks for these operations in linear algebra software libraries.

LAPACK (*Linear Algebra PACKage*) is a Fortran library of subroutines for solving many different numerical linear algebra problems, created in order to improve upon existing software libraries by taking better advantage of the memory hierarchies on modern machines. LAPACK routines do this by utilizing block matrix operations to a large extent. They are designed to use calls to the BLAS as extensively as possible, especially Level 3 BLAS.

Appendix C

Code

C.1 Block Cholesky factorization

```
/*
Block Cholesky factorization with StarPU.
Authors: Thomas McSweeney and Mawussi Zounon.
*/

#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<time.h>
#include<mkl.h>
#include<starpu.h>

// Useful function for printing matrices for visual
   comparison.
// Only really practical for very small sizes (n = 10 or
   less).
void print_matrix(double A[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%6.3f     ", A[j * n + i]);
        }
        printf("\n");
    }
}

// StarPU performance model.
static struct starpu_perfmodel pmodel = {
  .type = STARPU_HISTORY_BASED,
  .symbol = "pmodel"
```

```
};

// dpotrf kernel.
void starpu_cpu_dpotrf(void *buffers[], void *cl_arg) {

  // Unpack A.
  double *A = (double *) STARPU_MATRIX_GET_PTR(buffers[0]);

  int n, ld;
  // Could use StarPU functions to get n and ld, but for
   consistency with other functions, passed them as
   arguments instead.
  //n = STARPU_MATRIX_GET_NX(buffers[0]);
  //ld = STARPU_MATRIX_GET_LD(buffers[0]);

  starpu_codelet_unpack_args(cl_arg, &n, &ld);

  int layout = LAPACK_COL_MAJOR;
  char uplo = 'L';

  // Call the kernel.
  LAPACKE_dpotrf(layout, uplo, n, A, ld);
}

// dgemm kernel.
void starpu_cpu_dgemm(void *buffers[], void *cl_arg) {

  double *A = (double*) STARPU_MATRIX_GET_PTR(buffers[0]);
  double *B = (double*) STARPU_MATRIX_GET_PTR(buffers[1]);
  double *C = (double*) STARPU_MATRIX_GET_PTR(buffers[2]);

  // Could use STARPU_MATRIX_GET_NX and
   STARPU_MATRIX_GET_LD instead.
  int m, k, n;
  int lda, ldb, ldc;
  double alpha, beta;
  starpu_codelet_unpack_args(cl_arg, &m, &n, &k, &alpha, &
   lda, &ldb, &beta,
               &ldc);

  CBLAS_TRANSPOSE transA = CblasNoTrans;
  CBLAS_TRANSPOSE transB = CblasNoTrans;

  // Call DGEMM.
  cblas_dgemm(CblasColMajor, transA, transB,
```

```
            m, n, k,
            alpha, A, lda,
            B, ldb,
            beta, C, ldc);
}

// dtrsm kernel.
void starpu_cpu_dtrsm(void* buffers[], void *cl_arg) {

  double *A = (double *) STARPU_MATRIX_GET_PTR(buffers[0]);
  double *B = (double *) STARPU_MATRIX_GET_PTR(buffers[1]);

  double alpha;
  int m, n, lda, ldb;
  CBLAS_UPLO uplo = CblasLower;
  CBLAS_SIDE side = CblasLeft;
  CBLAS_DIAG diag = CblasNonUnit;

  CBLAS_LAYOUT layout = CblasColMajor;
  CBLAS_TRANSPOSE transA = CblasNoTrans;

  starpu_codelet_unpack_args(cl_arg, &m, &n,
                             &alpha, &lda, &ldb);

  cblas_dtrsm(layout, side, uplo, transA, diag,
            m, n, alpha, A, lda, B, ldb);
}

void starpu_cpu_dsyrk(void* buffers[], void *cl_arg) {

  double *A = (double *) STARPU_MATRIX_GET_PTR(buffers[0]);
  double *C = (double *) STARPU_MATRIX_GET_PTR(buffers[1]);

  double alpha, beta;
  int n, k, lda, ldc;
  CBLAS_UPLO uplo = CblasLower;
  CBLAS_LAYOUT layout = CblasColMajor;
  CBLAS_TRANSPOSE trans = CblasNoTrans;

  starpu_codelet_unpack_args(cl_arg, &n, &k,
                             &alpha, &lda, &beta, &ldc);

  cblas_dsyrk(layout, uplo, trans,
             n, k, alpha, A, lda, beta, C, ldc);
}
```

```c
// dpotrf codelet.
struct starpu_codelet starpu_codelet_dpotrf = {
  .cpu_func = starpu_cpu_dpotrf,
  .nbuffers = 1,
  .name = "dpotrf",
  .model = &pmodel
};

// dtrsm codelet.
struct starpu_codelet starpu_codelet_dtrsm = {
  .cpu_func = starpu_cpu_dtrsm,
  .nbuffers = 2,
  .name = "dtrsm",
  .model = &pmodel
};

// dgemm codelet.
struct starpu_codelet starpu_codelet_dgemm = {
  .cpu_func = starpu_cpu_dgemm,
  .nbuffers = 3,
  .name = "dgemm",
  .model = &pmodel
};

// dsyrk codelet.
struct starpu_codelet starpu_codelet_dsyrk = {
  .cpu_func = starpu_cpu_dsyrk,
  .nbuffers = 2,
  .name = "dsyrk",
  .model = &pmodel
};

// dpotrf task insertion.
static void starpu_dpotrf(int n, starpu_data_handle_t A,
   int lda) {

  starpu_insert_task(
           &starpu_codelet_dpotrf,
           STARPU_VALUE, &n, sizeof(int),
           STARPU_RW, A,
           STARPU_VALUE, &lda, sizeof(int),
           0);
}
```

```
// dpotrf task insertion.
static void starpu_dgemm(int m, int n, int k, double alpha,
    starpu_data_handle_t A, int lda, starpu_data_handle_t B
   , int ldb, double beta,
        starpu_data_handle_t C, int ldc) {

  starpu_insert_task(
            &starpu_codelet_dgemm,
            STARPU_VALUE, &m, sizeof(int),
            STARPU_VALUE, &n, sizeof(int),
            STARPU_VALUE, &k, sizeof(int),
            STARPU_VALUE, &alpha, sizeof(double),
            STARPU_R, A,
            STARPU_VALUE, &lda, sizeof(int),
            STARPU_R, B,
            STARPU_VALUE, &ldb, sizeof(int),
            STARPU_VALUE, &beta, sizeof(double),
            STARPU_RW, C,
            STARPU_VALUE, &ldc, sizeof(int),
            0);
}

// dsyrk task insertion.
static void starpu_dsyrk( int n, int k, double alpha,
   starpu_data_handle_t A, int lda, double beta,
   starpu_data_handle_t C, int ldc) {

  starpu_insert_task(
            &starpu_codelet_dsyrk,
            STARPU_VALUE, &n, sizeof(int),
            STARPU_VALUE, &k, sizeof(int),
            STARPU_VALUE, &alpha, sizeof(double),
            STARPU_R, A,
            STARPU_VALUE, &lda, sizeof(int),
            STARPU_VALUE, &beta, sizeof(double),
            STARPU_RW, C,
            STARPU_VALUE, &ldc, sizeof(int),
            0);
}

// dtrsm task insertion.
static void starpu_dtrsm(int m, int n, double alpha,
   starpu_data_handle_t A, int lda,  starpu_data_handle_t C
   , int ldc) {
  starpu_insert_task(
```

```
              &starpu_codelet_dtrsm,
              STARPU_VALUE, &m, sizeof(int),
              STARPU_VALUE, &n, sizeof(int),
              STARPU_VALUE, &alpha, sizeof(double),
              STARPU_R, A,
              STARPU_VALUE, &lda, sizeof(int),
              STARPU_RW, C,
              STARPU_VALUE, &ldc, sizeof(int),
              0);

}

int main(int argc, char *argv[]) {

    int check = starpu_init(NULL);
    STARPU_CHECK_RETURN_VALUE(check, "starpu_init");

    const int nb = 256;     // Block size.
    int n = 4096;           // Matrix size.
    int lda = n;            // Leading dimension.
    int nt = n / nb;        // Number of tiles (in the x and
    y directions).

    double *A = (double*) calloc(n * lda, sizeof(double));
    double *original_A = (double*) calloc(n * lda, sizeof(
   double));

    srand(time(NULL));      // Set a seed.

    // Initialize the matrix.
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++){
    A[i + j * n] = (double)rand()/RAND_MAX;
    if (i == j)
      A[i + j *n] += i*j + 1;  // Ensure that the matrix is
    diagonally dominant (and therefore positive definite)
      }
    }
    // Note matrix is not symmetric, but we only work with
   lower or upper half anyway.

    // Copy the matrix - useful for debugging and comparing
    relative function performance.
    memcpy(original_A, A, sizeof(double) * n * lda);
```

```
 // Register the handles.
 starpu_data_handle_t A_handles[nt * nt];
 for (int i = 0; i < nt; i++) {
   for (int j = 0; j < nt; j++) {
 starpu_matrix_data_register(&A_handles[i + j * nt],
STARPU_MAIN_RAM,
                 (uintptr_t)&A[i * nb + lda * j * nb],
lda, nb, nb, sizeof(double));
   }
 }

 clock_t start, time_taken;
 double cpu_time ,cpu_time_2;

 start = clock(); // Time StarPU function.

 // Factorize the matrix with StarPU.
 for (int k = 0; k < nt; k++) {
   starpu_dpotrf(nb, A_handles[k + k * nt], lda);
   for (int m = k + 1; m < nt; m++) {
 starpu_dtrsm(nb, nb, 1.0, A_handles[k + k * nt],
          lda, A_handles[m + k * nt], lda);
   }
   for (int m = k + 1; m < nt; m++) {
 starpu_dsyrk(nb,  nb, -1.0, A_handles[m + k * nt],
          lda, 1.0, A_handles[m +  m * nt], lda);
 for (int r = k + 1; r < nt; r++) {
   starpu_dgemm(nb, nb, nb, -1.0, A_handles[m + k * nt],
 lda,
           A_handles[r + k * nt], lda, 1.0, A_handles[m
 +  r * nt], lda);
 }
   }
 }

 // Wait for all tasks to be completed.
 starpu_task_wait_for_all();

 // Unregister the matrix.
 for (int i = 0; i < nt * nt; i++) {
   starpu_data_unregister(A_handles[i]);
 }

 time_taken = clock() - start;
 cpu_time = ((double) time_taken) / CLOCKS_PER_SEC;
```

```
  int layout = LAPACK_COL_MAJOR;
  char uplo = 'L';

  start = clock(); // Time MKL function.

  LAPACKE_dpotrf(layout, uplo, n, original_A, lda);

  time_taken = clock() - start;
  cpu_time_2 = ((double) time_taken) / CLOCKS_PER_SEC;

  // Now we compute the element-wise error - remember
 overwrote only the lower (or upper) triangular part.
  double error = 0.0;
  for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
  error += abs(A[i*n + j] - original_A[i*n + j]);
    }
  }

  printf("Performing the factorization with StarPU took %
 f seconds with error = %f.\n", cpu_time, error);
  printf("Factorizing the whole matrix with MKL took %f
 seconds.\n", cpu_time_2);

  // Shutdown StarPU.
  starpu_shutdown();
  return 0;
}
```

References

[1] Ahmad Abdelfattah, Hartwig Anzt, Aurelien Bouteiller, Anthony Danalis, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, Stephen Wood, Panruo Wu, Ichitaro Yamazaki, Asim Yarkhan, Jakub Luszczek, and Yarkhan Asim. Roadmap for the development of a linear algebra library for exascale computing: SLATE: Software for Linear Algebra Targeting Exascale, July 2017.

[2] Amit Agarwal and Padam Kumar. Economical duplication based task scheduling for heterogeneous and homogeneous computing systems. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, IEEE, 2009, pages 87–93.

[3] 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.

[4] Ishfaq Ahmad and Yu-Kwong Kwok. A new approach to scheduling parallel programs using task duplication. In *Parallel Processing, 1994. ICPP 1994 Volume 2. International Conference on*, volume 2, IEEE, 1994, pages 47–51.

[5] 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.

[6] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, 2002.

[7] 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.

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

[9] Haldun Aytug, Siddhartha Bhattacharyya, Gary J. Koehler, and Jane L. Snowdon. A review of machine learning in scheduling. *IEEE Transactions on Engineering Management*, 41(2):165–171, 1994.

[10] Savina Bansal, Padam Kumar, and Kuldip Singh. An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(6): 533–544, 2003.

[11] Blaise Barney. Message Passing Interface (MPI). [Online; accessed 10-June-2018].

[12] R. Bellman. *Dynamic Programming*. Dover Books on Computer Science. Dover Publications, 2013. ISBN 9780486317199.

[13] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[14] 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.

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

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

[17] Luiz F. Bittencourt, Rizos Sakellariou, and Edmundo R. M. Madeira. DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, IEEE, 2010, pages 27–34.

[18] Laurent Bobelin, Arnaud Legrand, Márquez Alejandro González David, Pierre Navarro, Martin Quinson, Frédéric Suter, and Christophe Thiery. Scalable Multi-Purpose Network Representation for Large Scale Distributed System Simulation. In *CCGrid 2012 – The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Ottawa, Canada, May 2012, page 19.

[19] Brent Bohnenstiehl, Aaron Stillmaker, Jon J. Pimentel, Timothy Andreas, Bin Liu, Anh T. Tran, Emmanuel Adeagbo, and Bevan M. Baas. KiloCore: A 32-nm 1000-Processor computational array. *IEEE Journal of Solid-State Circuits*, 52(4):891–902, 2017.

[20] 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.

[21] 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.

[22] Tom Carchrae and J. Christopher Beck. Applying machine learning to low-knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):372–387, 2005.

[23] 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.

[24] BSC-CNS Barcelona Supercomputing Center. OmpSs. [Online; accessed 27-May-2018].

[25] 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.

[26] 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.

[27] Y.-C. Chung and S. Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, Los Alamitos, CA, USA, 1992, pages 512–521. IEEE Computer Society Press.

[28] Ludovic Courtès. C language extensions for hybrid CPU/GPU programming with StarPU. *CoRR*, abs/1304.0878, 2013.

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

[30] Mohammad I. Daoud and Nawwaf Kharma. Efficient compile-time task scheduling for heterogeneous distributed computing systems. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, IEEE, 2006, pages 11–20.

[31] Mohammad I. Daoud and Nawwaf Kharma. A hybrid heuristic-genetic algorithm for task scheduling in heterogeneous processor networks. *Journal of Parallel and Distributed Computing*, 71(11):1518 – 1531, 2011.

[32] 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.

[33] Marc Deisenroth and Carl E. Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, 2011, pages 465–472.

[34] Marc Peter Deisenroth, Dieter Fox, and Carl Edward Rasmussen. Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(2):408–423, 2015.

[35] The ICL Distributed Computing Group. DPLASMA. [Online; accessed 12-June-2018].

[36] The ICL Distributed Computing Group. MAGMA. [Online; accessed 12-June-2018].

[37] The ICL Distributed Computing Group. PLASMA. [Online; accessed 12-June-2018].

[38] Hesham El-Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138 – 153, 1990.

[39] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, IEEE, 2011, pages 365–376.

[40] Marc-Eduard Frincu, Martin Quinson, and Frédéric Suter. Handling Very Large Platforms with the New SimGrid Platform Description Formalism. Technical Report RT-0348, INRIA, 2008. 27 pp.

[41] 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.

[42] Jie Gao, Linyan Sun, and Mitsuo Gen. A hybrid genetic and variable neighborhood descent algorithm for flexible job shop scheduling problems. *Computers and Operations Research*, 35(9):2892 – 2907, 2008. Part Special Issue: Bio-inspired Methods in Combinatorial Optimization.

[43] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.*, 1(2):117–129, 1976.

[44] 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.

[45] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[46] Green500. November 2017. [Online; accessed 14-May-2018].

[47] 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.

[48] Mourad Hakem and Franck Butelle. Dynamic critical path scheduling parallel programs onto multiprocessors. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, IEEE, 2005, pages 7–pp.

[49] M. R. Hilliard, G. E. Liepins, and M. Palmer. Machine learning applications to job shop scheduling. In *Proceedings of the 1st International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems - Volume 2*, IEA/AIE '88, New York, NY, USA, 1988, pages 728–737. ACM.

[50] E. S. H. Hou, N. Ansari, and Hong Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, 1994.

[51] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D. Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[52] Intel. Intel Math Kernel Library. [Online; accessed 22-May-2018].

[53] 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.

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

[55] 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.

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

[57] Ali Husseinzadeh Kashan, Behrooz Karimi, and Masoud Jenabi. A hybrid genetic heuristic for scheduling parallel batch processing machines with arbitrary job sizes. *Computers and Operations Research*, 35(4):1084 – 1098, 2008.

[58] 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.

[59] The Khronos Group. OpenCL. [Online; accessed 12-June-2018].

[60] Marina Krstic Marinkovic and Luka Stanisic. Platform independent profiling of a QCD code. In *Lattice 2016 - 34th annual International Symposium on Lattice Field Theory*, PoS, Southampton, United Kingdom, July 2016, pages 1–7.

[61] Boontee Kruatrachue and Ted Lewis. Grain size determination for parallel processing. *IEEE software*, 5(1):23–32, 1988.

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

[63] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, 7(5):506–521, 1996.

[64] 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.

[65] C.-Y. Lee, S. Piramuthu, and Y.-K. Tsai. Job shop scheduling with a genetic algorithm and machine learning. *International Journal of Production Research*, 35(4):1171–1191, 1997.

[66] Arnaud Legrand. *Scheduling for Large Scale Distributed Computing Systems: Approaches and Performance Evaluation Issues*. Habilitation à diriger des recherches, Université Grenoble Alpes, November 2015.

[67] 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.

[68] Kenli Li, Xiaoyong Tang, Bharadwaj Veeravalli, and Keqin Li. Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems. *IEEE Transactions on computers*, 64(1):191–204, 2015.

[69] Kenli Li, Zhimin Zhang, Yuming Xu, Bo Gao, and Ligang He. Chemical reaction optimization for heterogeneous computing environments. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, IEEE, 2012, pages 17–23.

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

[71] Jing-Chiou Liou and Michael A. Palis. An efficient task clustering heuristic for scheduling DAGs on multiprocessors. In *Workshop on Resource Management, Symposium on Parallel and Distributed Processing*, 1996, pages 152–156.

[72] Bo Liu, Ling Wang, and Yi-Hui Jin. An effective hybrid PSO-based algorithm for flow shop scheduling with limited buffers. *Computers and Operations Research*, 35(9):2791 – 2806, 2008. Part Special Issue: Bio-inspired Methods in Combinatorial Optimization.

[73] Chun-Hsien Liu, Chia-Feng Li, Kuan-Chou Lai, and Chao-Chin Wu. A dynamic critical path duplication task scheduling algorithm for distributed heterogeneous computing systems. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, IEEE, 2006, pages 365–373.

[74] DeepMind Technologies Ltd. DeepMind. [Online; accessed 26-June-2018].

[75] Gary Marcus. Deep learning: A critical appraisal. *CoRR*, abs/1801.00631, 2018.

[76] Microsoft. Project Catapult. [Online; accessed 14-May-2018].

[77] Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

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

[79] 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.

[80] Lars Mönch, Jens Zimmermann, and Peter Otto. Machine learning techniques for scheduling jobs with incompatible families and unequal ready times on parallel batch machines. *Engineering Applications of Artificial Intelligence*, 19(3):235 – 245, 2006.

[81] Rebecca Morelle. Google machine learns to master video games. [Online; accessed 26-June-2018].

[82] 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.

[83] 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.

[84] 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.

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

[86] NLAFET. Github. [Online; accessed 18-May-2018].

[87] NLAFET. Parallel numerical linear algebra for extreme scale systems. [Online; accessed 18-May-2018].

[88] J. R. Norris. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997.

[89] OpenMP. The OpenMP API specification for parallel programming. [Online; accessed 17-May-2018].

[90] 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.

[91] 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.

[92] Andrew J. Page, Thomas M. Keane, and Thomas J. Naughton. Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system. *Journal of Parallel and Distributed Computing*, 70(7):758 – 766, 2010.

[93] Andrew J. Page and Thomas J. Naughton. Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, IEEE, 2005, pages 8–pp.

[94] 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.

[95] Byung Joo Park, Hyung Rim Choi, and Hyun Soo Kim. A hybrid genetic algorithm for the job shop scheduling problems. *Computers and Industrial Engineering*, 45(4):597 – 613, 2003.

[96] Gyung-Leen Park, Behrooz Shirazi, and Jeff Marquis. DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor systems. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, IEEE, 1997, pages 157–166.

[97] Steven Pinker. *The Blank Slate: The Modern Denial of Human Nature*. Penguin, 2003.

[98] 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.

[99] Paolo Priore, David de la Fuente, Javier Puente, and José Parreño. A comparison of machine-learning algorithms for dynamic scheduling of flexible manufacturing systems. *Engineering Applications of Artificial Intelligence*, 19(3):247 – 255, 2006.

[100] Martin Quinson. *Computational Science of Computer Systems*. Habilitation à diriger des recherches, Université de Lorraine, March 2013.

[101] Martin Quinson, Cristian Rosa, and Christophe Thiery. Parallel Simulation of Peer-to-Peer Systems. In *CCGrid 2012 – The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12 Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Ottawa, Canada, May 2011, pages 668–675. IEEE.

[102] 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.

[103] Hanan Rosemarin, John P. Dickerson, and Sarit Kraus. Learning to schedule deadline- and operator-sensitive tasks. *CoRR*, abs/1706.06051, 2017.

[104] S. Ruder. An overview of gradient descent optimization algorithms. *ArXiv e-prints*, 2016.

[105] Daniel Russo, Benjamin Van Roy, Abbas Kazerouni, and Ian Osband. A tutorial on Thompson sampling. *CoRR*, abs/1707.02038, 2017.

[106] 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.

[107] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[108] A. L. Samuel. Some studies in machine learning using the game of checkers. II—Recent progress. *IBM Journal of Research and Development*, 11(6): 601–617, 1967.

[109] M. Santos and J. Rust. Convergence properties of policy iteration. *SIAM Journal on Control and Optimization*, 42(6):2094–2115, 2004.

[110] 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.

[111] Nakasuka Schinichi and Yoshida Taketoshi. Dynamic scheduling system utilizing machine learning as a knowledge acquisition tool. *International Journal of Production Research*, 30(2):411–431, 1992.

[112] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61(Supplement C):85 – 117, 2015.

[113] M. J. Shaw, S. Park, and N. Raman. Intelligent scheduling with machine learning capabilities: The induction of scheduling knowledge. *IIE Transactions*, 24(2):156–168, 1992.

[114] Gilbert C. Sih and Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE transactions on Parallel and Distributed systems*, 4(2):175–187, 1993.

[115] 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.

[116] 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.

[117] SimGrid. Describing the virtual platform. [Online; accessed 18-June-2018].

[118] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[119] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. Fast and accurate simulation of multithreaded sparse linear algebra solvers. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2015, pages 481–490.

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

[121] 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.

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

[123] Dave Steinkraus, I. Buck, and P.Y. Simard. Using GPUs for machine learning algorithms. In *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on*, IEEE, 2005, pages 1115–1120.

[124] 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.

[125] Frédéric Suter. DAGGEN. [Online; accessed 19-June-2018].

[126] Frédéric Suter. *Bridging a Gap Between Research and Production: Contributions to Scheduling and Simulation*. Habilitation à diriger des recherches, Ecole normale supérieure de Lyon, December 2014.

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

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

[129] Gerald Tesauro. Practical issues in temporal difference learning. In *Advances in neural information processing systems*, 1992, pages 259–266.

[130] Gerald Tesauro. TD-Gammon: A self-teaching backgammon program. In *Applications of Neural Networks*, Alan F. Murray, editor, Boston, MA, 1995, pages 267–285. Springer US.

[131] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[132] Gerald Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134(1):181 – 199, 2002.

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

[134] William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25 (3/4):285–294, 1933.

[135] William R. Thompson. On the theory of apportionment. *American Journal of Mathematics*, 57(2):450–456, 1935.

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

[137] Top500. November 2017. [Online; accessed 08-May-2018].

[138] 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.

[139] 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.

[140] John Tromp. Number of legal Go positions. [Online; accessed 26-June-2018].

[141] John N. Tsitsiklis. On the convergence of optimistic policy iteration. *Journal of Machine Learning Research*, 3(Jul):59–72, 2002.

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

[143] Pedro Velho and Arnaud Legrand. Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. In *SIMUTools'09, 2nd International Conference on Simulation Tools and Techniques*, Rome, Italy, March 2009.

[144] 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.

[145] 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.

[146] Chris Walker. New Intel core processor combines high-performance CPU with custom discrete graphics from AMD to enable sleeker, thinner devices. [Online; accessed 16-May-2018].

[147] Lee Wang, Howard Jay Siegel, Vwani P. Roychowdhury, and Anthony A. Maciejewski. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of Parallel and Distributed Computing*, 47(1):8 – 22, 1997.

[148] 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.

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

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

[151] Yun Wen, Hua Xu, and Jiadong Yang. A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system. *Information Sciences*, 181(3):567 – 581, 2011.

[152] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4 (2):65–85, 1994.

[153] Wikichip. PEZY-SC2. [Online; accessed 16-May-2018].

[154] Wikipedia. Blue Gene. [Online; accessed 15-May-2018].

[155] 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.

[156] 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.

[157] Yuming Xu, Kenli Li, Ligang He, Longxin Zhang, and Keqin Li. A hybrid chemical reaction optimization scheme for task scheduling on heterogeneous computing systems. *IEEE Transactions on parallel and distributed systems*, 26(12):3208–3222, 2015.

[158] Yuming Xu, Kenli Li, Jingtong Hu, and Keqin Li. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Information Sciences*, 270(Supplement C):255 – 287, 2014.

[159] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[160] Afshin Zafari, Elisabeth Larsson, and Martin Tillenius. DuctTeip: An efficient programming model for distributed task based parallel computing. *CoRR*, abs/1801.03578, 2018.

[161] Fan Zhang, Junwei Cao, Keqin Li, Samee U. Khan, and Kai Hwang. Multi-objective scheduling of many tasks in cloud platforms. *Future Generation Computer Systems*, 37(Supplement C):309 – 320, 2014. Special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications.

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

[163] Ziliang Zong, Adam Manzanares, Xiaojun Ruan, and Xiao Qin. EAD and PEBD: two energy-aware duplication scheduling algorithms for parallel tasks on homogeneous clusters. *IEEE Transactions on Computers*, 60 (3):360–374, 2011.

[164] Mawussi Zounon. Novel methods for static and dynamic scheduling. [Online; accessed 18-May-2018].