SCHEDULING WITH PRECEDENCE CONSTRAINTS IN HETEROGENEOUS PARALLEL COMPUTING

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2021

Thomas E. McSweeney
Department of Mathematics
School of Natural Sciences

Table of Contents

Word count 55800

List of Tables

List of Figures

The University of Manchester

Thomas E. McSweeney
Doctor of Philosophy
Scheduling with Precedence Constraints in Heterogeneous Parallel Computing
December 6, 2021

Modern computers are parallel. From the most powerful supercomputers to desktop machines, computing environments with multiple processing resources are ubiquitous. Moreover, these resources are increasingly likely to be heterogeneous, differing widely from one another in terms of their speed, energy consumption and so on. To exploit this new landscape, scientific computing applications are often expressed in the form of a graph, with vertices representing discrete chunks of work called tasks and edges indicating the order in which they must be executed. This exposes the parallelism of the application, but provokes an immediate question: which tasks should each processor execute, and when? In other words, how do we find the optimal schedule that the processors should follow?

First, we consider the case when there are only two different types of processing resources. Computing environments of this type are widespread today, most commonly comprising multicore CPUs and one or more GPUs. However many of the algorithms used for scheduling such platforms do not exploit their unique properties. We investigate how a common heuristic framework in which tasks are assigned priorities and scheduled in this order can be optimized for accelerated platforms. We propose a suite of possibilities and compare their performance with existing methods through extensive simulation, finding that different choices are better in different situations.

Key to many scheduling heuristics is anticipating the critical path, the longest sequence of tasks during the schedule execution. We describe how approximations of the critical path are computed and used in scheduling heuristics for generic heterogeneous platforms, suggesting also new methods of our own. These are then evaluated through simulation in order to establish whether they are useful and, if so, when.

Given a schedule, how long will it actually take? This is a trivial question when the task execution times and communication delays can be predicted exactly. But in practice that will never be true, so they are often modeled as random variables instead of scalars. However, computing the schedule duration then becomes an intractable problem. We suggest a heuristic framework that may be useful for computing approximate solutions and compare it to existing algorithms through numerical experimentation.

We conclude by considering the natural next question: how do we compute a schedule which is robust to variation in the task execution and communication times? After reviewing existing heuristics, with a focus on those that work by first transforming the problem into a deterministic one, we propose a modification to one such algorithm which appears to improve its rate of convergence.

Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

ACKNOWLEDGMENTS

I would like to thank my supervisor, Neil Walton, for all the support he has provided over the last few years, and especially for convincing me that it's not the end of the world when things don't work. I will always be grateful. Enormous credit is also due to Mawussi Zounon for reminding me to look at the big picture and always being willing to help. I should also thank my office mates and the other postgrads who called the Alan Turing building home (until the pandemic came along!) for all the much-needed distractions.

Finally, I never could have started this, let alone finished it, if it wasn't for the love and support of my family, so they really deserve all the credit, especially my mother. Thanks, Mum.

Chapter 1

Introduction

The most powerful modern supercomputers are fast approaching *exascale*, the capability to perform at least 10^{18} floating-point operations per second. This represents orders of magnitude more computational power than ever before. Two hardware trends are largely responsible: supercomputers are now *massively parallel*, comprising thousands or even millions of processing resources, and it is increasingly likely that these resources will be *heterogeneous*. This architectural shift offers awesome potential: ideally, all of the constituent parts of complex scientific computing applications will be performed on the resource types that minimize their processing time, leading to almost perfect efficiency. But realizing this goal requires successfully coordinating large sets of diverse processing resources, allocating work to each of the processors so that the application runtime is minimized and we avoid undue communication delays. This means that exploiting modern high performance computing (HPC) architectures to their fullest extent is only possible if we can devise good *schedules* for their many, possibly heterogeneous, processors.

In this thesis, we study various permutations of this vitally important scheduling problem. Efficiency is paramount in HPC, where machines can have annual energy costs in the millions of dollars, so that is where the problem is most pertinent. But, as the current hardware trends in HPC inevitably filter downward to everyday computing, we expect the problems studied here to become more widely relevant in the immediate future.

1.1 TASKS AND GRAPHS

Although high-performance computers have been almost universally parallel since at least the 1990s, the massive recent growth in the number of processing resources offers much more scope for parallel computation than ever before. Increasingly, machines have a hierarchy of parallelism: across different compute nodes, between the processors within those nodes, among the cores of those processors, and so on. This means that now almost all codes that we write must be parallelizable in order to take full of advantage of the available resources. This is problematic in light of the fact that parallel computing has traditionally proven difficult from a programming perspective [44]. Therefore, the question is, how do we successfully exploit all of this new parallelism without departing too radically from the existing programs and software that have proven to be successful?

One paradigm that has become popular in recent years for striking this balance is *task-based* parallel programming. The basic idea is to express the application that we wish to execute as a collection of logically discrete atomic units of work called *tasks* and specify the *precedence constraints* (or *dependencies*) between them. This gives us a *task (dependency) graph*, where the vertices represent the tasks and the edges the precedence constraints. Some applications naturally comprise sets of tasks, whereas for others there may be flexibility in how tasks are defined. However, in this research we consider only applications whose task graphs are *directed acyclic graphs* (DAGs)—directed and without any cycles. Many scientific computing applications can be expressed in this form, as illustrated by the example below. Writing parallel programs becomes much easier in the task-based framework: the programmer just needs to implement sequential code—or use efficient existing code—that operates at the task level. The downside is that good performance now depends to a large extent on the ability to construct good schedules for the resulting task graphs.

1.1.1 Example

In numerical linear algebra (NLA), *Basic Linear Algebra Subprograms* (BLAS) [50] is a specification for extremely efficient, portable routines that perform fundamental 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 libraries, such as the classic *Linear Algebra PACKage* (LAPACK) [11], which is designed to use calls to the BLAS as extensively as possible. BLAS and LAPACK implementations are extremely common and vendors often provide highly-optimized versions for their own architectures, such as the *Math Kernel Library* (MKL) [62] for Intel processors, and the cuBLAS [94] and cuSOLVER [95] libraries for NVIDIA GPUs.

As an example of how the BLAS may be used in the task-based programming framework, suppose that we have a large, symmetric, positive-definite $N \times N$ tiled matrix A,

$$A = \begin{bmatrix} A_{11} & \dots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} & \dots & A_{NN} \end{bmatrix},$$

and we wish to compute its Cholesky factorization—i.e., find a lower triangular matrix L such that $A = LL^T$. There is an LAPACK kernel for computing the Cholesky factorization, namely `POTRF`, that we could in theory apply directly to the entire matrix. However, this does not exploit the parallelism inherent in the factorization itself. Superior performance can be achieved by using `POTRF` at the tile level, in conjunction with common BLAS routines, in order to compute the Cholesky factorization of A in a manner which exposes the underlying parallelism. Algorithm 1.1 outlines a standard way that this may be implemented using `GEMM` (matrix multiplication), `SYRK` (symmetric rank-k update) and `TRSM` (triangular solve) BLAS kernels [64], [77]. Note that, for the sake of efficiency, the lower triangular part of the matrix A is overwritten with the entries of L.

By defining tasks as kernel calls on individual tiles of the matrix, it is straightforward to construct the topology of a task graph which corresponds to Algorithm 1.1 once N, the number of tiles along each axis, has been specified; for example, Figure 1.1 shows the DAG topology when $N = 5$. Moreover, there is nothing special about Cholesky factorization: a wide variety of other applications can also be expressed in the form of a task DAG, both in NLA and across scientific computing in general.

1.1.2 Notation

At this point, we introduce some graph-related notation and terminology which will be used throughout this thesis. A *task graph* G comprises n tasks and v edges. Tasks are denoted by t_i, for $i = 1, \dots, n$, and are assumed to be labeled in such a way that for any

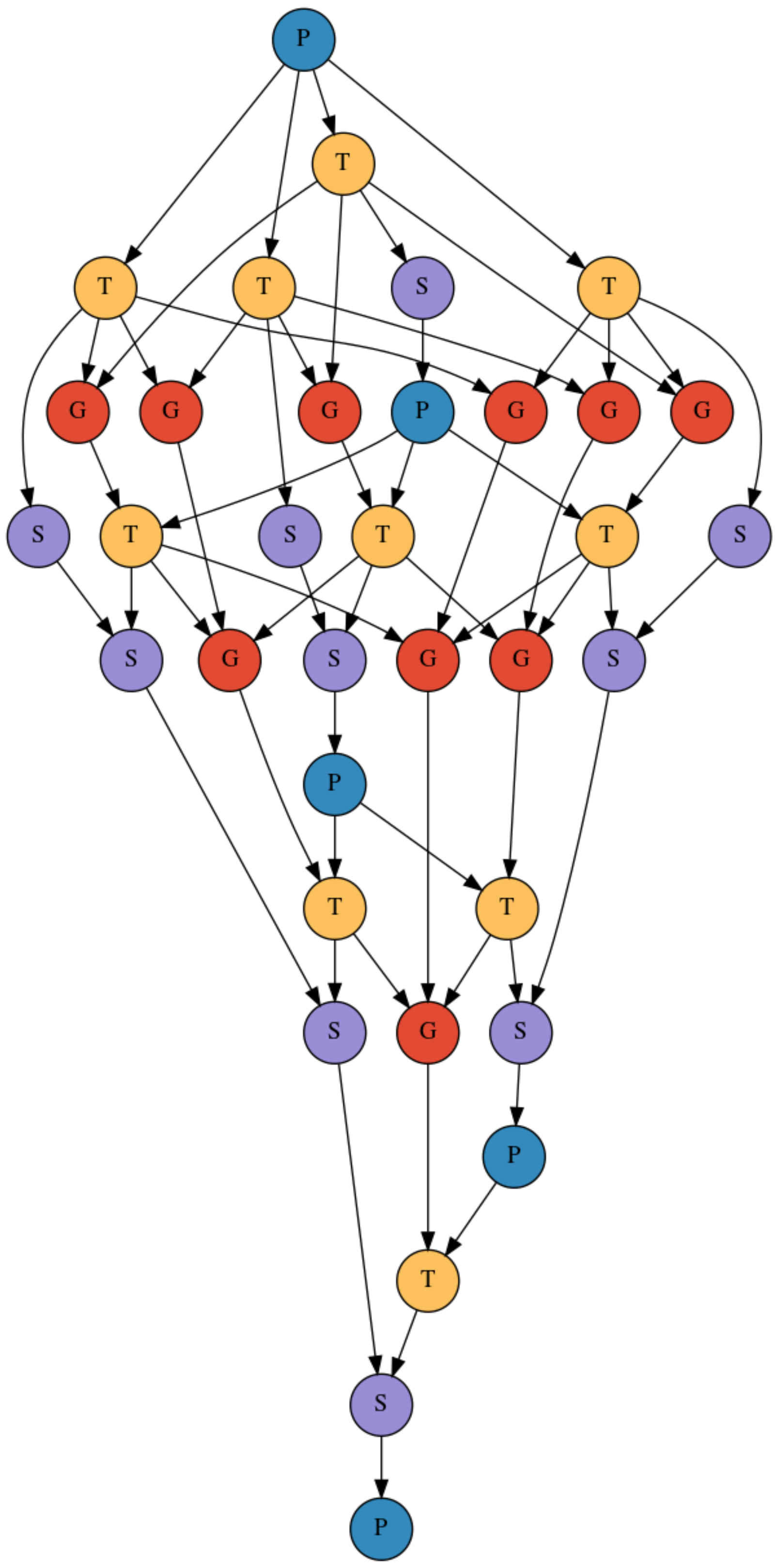


Figure 1.1: Task DAG for Cholesky factorization of a 5 × 5 tiled matrix. Labels and colors indicate the BLAS/LAPACK routine that the task represents. Tile indices not shown for clarity.

Algorithm 1.1: A practical algorithm for computing the Cholesky factorization of a matrix using BLAS/LAPACK kernels.

```
for $i = 1, \dots, N$ do
    $A_{ii}$ = POTRF($A_{ii}$)
    for $j = i+1, \dots, N-1$ do
        $A_{ji}$ = TRSM($A_{ii}, A_{ji}$)
    end
    for $k = i+1, \dots, N-1$ do
        $A_{kk}$ = SYRK($A_{ki}, A_{kk}$)
        for $j = k+1, \dots, N-1$ do
            $A_{jk}$ = GEMM($A_{ji}, A_{ki}, A_{jk}$)
        end
    end
end
```

Table 1.1: Common notation used throughout thesis.

G	Task graph
n	Number of tasks
v	Number of edges
t_i	An individual task, indexed by $i = 1, \dots, n$
t_1	The entry task/source
t_n	The exit task/sink
(t_i, t_k)	An edge from t_i to t_k
Γ_i^+	Indices of children of t_i, $k \in \Gamma_i^+ \iff \exists\,(t_i, t_k)$
Γ_i^-	Indices of parents of t_i, $h \in \Gamma_i^- \iff \exists\,(t_h, t_i)$

directed edge (t_i, t_k) from t_i to t_k we have $k > i$; in other words, $t_1, t_2, \dots, t_n$ is a *topological sort* of the tasks. We refer to the immediate successors of a task as its *children* and define Γ_i^+ to be the set of indices of the children of t_i—i.e., $k \in \Gamma_i^+ \iff \exists\,(t_i, t_k)$. Similarly, the immediate predecessors of a task are referred to as its *parents*, whose indices are contained in the set Γ_i^-. A task with no parents is called an *entry* task or *source*, and one with no children is an *exit* task or *sink*. The DAG in Figure 1.1 has only one source and sink. In principle there may be many, however in such cases it is often convenient to add a single artificial source, representing a dummy task, and likewise for the sink. Therefore, without loss of generality, we will assume that all of the task graphs we consider in this thesis have only one source t_1 and one sink t_n. Table 1.1 summarizes all of this notation for future reference.

1.2 Scheduling

Given a heterogeneous target platform and a task DAG that we wish to execute on it, the immediate question that arises is: how do we assign the tasks to the processing resources in the best possible way? In other words, what *schedule* should we follow? A schedule is a mapping from tasks to processing resources that tells us which tasks should be executed by each processor and, ideally, in what order this should be done and precisely when each task's execution should begin. Any prospective schedule should be *valid* in the sense that precedence constraints are respected and no task's execution is attempted until all of its children have been processed. But, of course, we don't just want any valid schedule; we want to find the schedule which optimizes one or more objectives. In HPC, this will typically be to minimize the *makespan*, the total length of time it takes to execute the schedule (i.e., the finish time of the exit task). Other objectives, such as minimizing the total energy expenditure are also clearly desirable as well. Objectives will always be clearly stated when we discuss specific problems in later chapters.

For a given objective function, we can define for each task and each processor a *cost* that encodes useful information about how scheduling the task on the processor contributes toward the objective. For example, if we want to minimize the makespan, it is sensible to define the cost of the task as its execution time on the processor. Similarly, we could define for each edge a set of costs corresponding to the communication delay between the relevant tasks when they are scheduled on different pairs of processors. In practice, these costs, however we choose to define them, will rarely be known exactly before runtime, when the tasks are actually processed and the necessary data transfers are made. However, we can model them beforehand, either as scalars (see Chapters 2 and 3) or random variables (Chapters 4 and 5).

A fundamental distinction is made between *offline* (or *static*) scheduling and *online* (or *dynamic*) scheduling. Static schedules are fixed before execution—i.e., computed offline—based on cost estimates available at that time, whereas dynamic schedules are determined during runtime. There are generic advantages and disadvantages to both: offline scheduling allows deeper analysis of the task graph so is typically superior when cost estimates are sufficiently accurate, whereas online scheduling is more practical when cost estimates are poor or only become apparent at runtime. We are mostly concerned with offline scheduling in this thesis but there will be references to online scheduling throughout.

As a general rule scheduling problems tend to be computationally hard and the ones considered here are not exceptions. Formal complexity results are stated where relevant in later chapters but it suffices to say here that the problems that we study are typically so difficult that we must rely on heuristics that give us reasonably good solutions with practical runtimes. Note that we will assume some familiarity with basic concepts from computational complexity theory, such as what it means for a problem to be *NP-hard* or the definition of an *approximation algorithm*.

1.3 STRUCTURE OF THESIS

The remainder of this thesis comprises five chapters. Other than Chapter 6, the conclusion, which summarizes all of the preceding chapters and outlines potential areas of future research, each chapter is focused on a separate topic that falls under the rubric of heterogeneous scheduling. We provide the following high level overview.

In Chapter 2, we consider the specific problem of scheduling for *accelerated* platforms comprising processing resources of only two different types, without loss of generality assumed to be CPUs and GPUs. Computing environments of this type have become ubiquitous in HPC, however many of the scheduling heuristics that are employed for them were originally developed for more diversely heterogeneous platforms and do not successfully exploit their binary heterogeneity. Therefore the aim of this chapter is to investigate how a common heuristic framework, in which tasks are assigned priorities then scheduled in this order according to some processor selection rule, can be optimized for accelerated architectures. In particular, tasks are often prioritized according to the estimated length of the longest (or *critical*) path from each task to the sink. But in heterogeneous computing these critical path lengths cannot be predicted during scheduling since there are many possible values each of the DAG weights may take. One solution is to *average* the possible weights and compute the critical paths of the resulting scalar-weighted graph. However, it isn't clear which type of average is most effective for CPU-GPU platforms. Our contribution in this thesis is to propose several averages that intuition suggests may be well-suited for such platforms and compare them experimentally to others from the literature, identifying which are the most useful and when. In addition, we introduce a generic method intended to improve any existing schedule by extracting the most accurate weights to use when computing critical path lengths as task priorities. Finally, we propose two simple new processor selection rules and compare their performance to several existing rules through simulation.

In Chapter 3, we extend part of the investigation conducted in Chapter 2 to generic—i.e., not just CPU and GPU—heterogeneous platforms and consider the broader question of how the critical path should be both approximated and used in scheduling heuristics. As alternatives to the averaging method for estimating the critical path, our main contribution is to introduce two new approaches, one based on computing bounds on the path length and the other a stochastic interpretation of the problem. We then evaluate their performance compared to multiple different average types experimentally.

Rather than focusing on how we can compute a good schedule, in Chapter 4 our objective is the difficult related problem of how we can efficiently approximate the makespan of a given schedule when the schedule costs are no longer scalars but stochastic. This is an instance of a more general problem, with applications beyond scheduling, in which the aim is to find the distribution of the longest path through a DAG with stochastic weights. Our contributions here are threefold. First, by generating empirical schedule makespan distributions for a real application, namely Cholesky factorization, we show that the common assumption that the distribution will be approximately normal may not hold. Next, we evaluate existing heuristics for the stochastic longest path problem through simulation, identifying their respective strengths and weaknesses. Finally, we propose a new heuristic framework of own, based on identifying a reasonably-sized subset of paths which are likely to be critical and approximating the maximization of their lengths, which aims to efficiently approximate the longest path distribution without assuming normality.

Tying together the preceding chapters, the aim of Chapter 5 is to investigate how we can compute a schedule which compensates for stochasticity in the cost estimates. Dealing with stochastic costs is difficult, so a popular approach to the problem is to convert them to scalars. In particular, one existing iterative algorithm works by repeatedly sampling the costs randomly from their distributions and computing candidate schedules from the corresponding deterministic problems. Although conceptually simple, this method may require many iterations to obtain a high-quality schedule. Therefore our main contribution in this chapter is to propose and evaluate an alternative method of scalarizing the costs with the goal of producing good candidate schedules more quickly.

1.4 REPRODUCIBILITY

Except for Chapter 6 and this introduction, in every other chapter of this thesis we utilize bespoke simulation software in order to evaluate new or existing algorithms. In the interest of reproducible research, all of this software can be found at the following Github repository:

`https://github.com/mcsweeney90/thesis-code`.

Moreover, the specific code relevant to each chapter can be found in the directory with the corresponding name—e.g., `chapter2` contains the codes for Chapter 2, and so on. All of the simulation software was written in `Python`, as were the scripts used to generate and analyze results. Specific version and package requirements are given in the repository so that readers can repeat the simulations, or modify the code, if they so wish. All of the results presented in this thesis were obtained on a desktop machine with an Intel i9-9820X @3.30 GHz processor running `Python 3.9.2` under an `Ubuntu 18.04` operating system.

CHAPTER 2

OPTIMIZING SCHEDULING HEURISTICS FOR ACCELERATED ARCHITECTURES

Modern heterogeneous computing environments often consist of just two kinds of processors: multicore CPUs and some type of *accelerator*, usually GPUs. However, many of the heuristics that are commonly used for scheduling such platforms were originally designed for much more diversely heterogeneous systems and do not successfully exploit this duality. Therefore, in this chapter we investigate how one popular heuristic framework that is known to perform well in the general case can be optimized specifically for accelerated platforms.

Note that this work is closely related to the paper "An efficient new static scheduling heuristic for accelerated architectures" [85] which was published in the proceedings of the International Conference on Computational Science (ICCS) 2020[1]. The conference paper introduces a new scheduling heuristic for accelerated architectures called *Heterogeneous Optimistic Finish Time* (HOFT). However, in this chapter we take a broader view and propose a suite of potential optimizations for the general heuristic framework that HOFT belongs to, in addition to those which were described in the paper. This chapter and the published paper should therefore be viewed as complementary.

[1] https://www.iccs-meeting.org/iccs2020/

2.1 RISE OF THE GPU

About fifteen years ago, it became apparent that although Moore's Law continued to hold true, the performance improvement of single-core processors had begun to stagnate. This was primarily because physical upper limits on their clock frequencies had been reached, a problem sometimes referred to as the *frequency wall* [13]. To overcome this hurdle, designers instead began to increase the number of processing cores on each individual chip; thus we entered the *multicore* era. Today, mainstream computers—desktops, laptops, tablets, games consoles—almost universally host multicore processors and modern HPC systems may comprise millions of cores in total. The Oak Ridge National Laboratory's Summit, currently second on the TOP500 [88] list of the world's fastest supercomputers, has about 2.4 million cores in total—and this is far from atypical.

Although the move towards multicore allowed processors to maintain constant performance improvement, energy issues soon came to the forefront. As processors have grown more and more powerful, their energy consumption and heat production has grown in kind, a problem sometimes called the *power wall* [37]. To alleviate this, *heterogeneous* machines with different kinds of processors have became increasingly attractive. The idea is that by performing vital (or compute-intensive) jobs on higher-power processors and less pressing (or compute-intensive) jobs on lower power ones, we can reduce wasted computational effort and improve overall energy efficiency [59]. This has proven to be a successful template; for example, nine of the current top ten on the Green500 [87] list of the world's most energy efficient supercomputers—measured in terms of *floating point operations* (flops) per watt of energy supplied—are of this type.

In the longer term it is expected that HPC architectures will comprise many different processor types, each specialized for various tasks [15], [28]. However, at present the landscape is dominated by platforms comprising multicore CPUs and more powerful *accelerators* of another type. A wide variety of devices have been employed in this fashion, such as *field-programmable gate arrays* (FPGAs), integrated chips that can be programmed for specialized tasks. However the real driver of this trend has been the *graphics processing unit* (GPU). For example, Summit comprises over 4000 nodes, each with two 22-core IBM Power9 CPUs and six NVIDIA Tesla V100 GPUs.

Spurred by their importance in the gaming industry, GPUs have become increasingly cheap and powerful in recent years. Moreover, being optimized for graphics rendering, they have proven to be adept at many other large, parallel applications [96]. Initially research was largely focused on how GPUs (and other accelerators) could be used as a replacement for the CPUs, or how work could be offloaded to them with little overall coordination. However, the consensus today is that in order to achieve the best possible performance we need to seamlessly integrate the different processor types [89]. In particular, this means that we require *schedules* for such platforms which successfully harness both types of processor.

2.2 SCHEDULING FOR CPU AND GPU

To reduce the burden on the programmer, in practice task scheduling on accelerated platforms is usually handled by a *runtime system* such as OpenMP [43], OmpSs [51], Quark [142], ParSeC [28], StarSs [98], or StarPU [15]. Typically, these are designed for online scheduling: all task allocations are decided at runtime based on the current state of the platform and the set of tasks ready to be processed [127]. There are good practical reasons for this decision. Shared resources make predicting task execution and data movement times on modern platforms extremely difficult [3]. Furthermore, processing the entire task DAG in the manner usually required by offline scheduling techniques can be prohibitively expensive; many modern systems such as PaRSEC unfurl the DAG at runtime and never allow users to view it in its entirety. Nevertheless, in this chapter we focus largely on offline scheduling, which would appear to be a contradiction. However, we believe it is justifiable to devote attention to the offline scheduling problem for the following reasons.

1. Good schedules can be surprisingly robust, even when the estimates used to compute them are poor [2], [33]. Furthermore, modifying an existing schedule may be advantageous compared to creating a new one from scratch at runtime [107].

2. Superior performance is possible in online scheduling contexts by using information gathered from the DAG offline to guide decision-making (e.g., task priorities) [2], [3], [37].

3. More broadly, the principles underlying successful offline heuristics can often be adapted for online scheduling. For example, multiple online versions of the classic *Heterogeneous Earliest Finish Time* (HEFT) [131] heuristic have been proposed and are actually used in real runtime systems [15], [37], [127].

4. Some runtime systems such as StarPU enable the offline analysis of application task graphs built from previous execution traces [53], [127]. Incorporating real data in such a way will improve the accuracy of the timing estimates used, making static schedules more practical.

With these justifications in mind, we define the offline scheduling model that we follow here more precisely in the next section. We refer the reader to the recent survey by Beaumont et al. [19] for a comprehensive guide to the current state-of-the-art in online DAG scheduling for accelerated platforms.

2.2.1 Model and definitions

Given an (accelerated) target platform and a task DAG representing an application that we wish to run on it, fundamentally the scheduling problem we wish to consider is: which tasks should be executed by each processing resource, and when? In this section we introduce the scheduling model that we follow in order to study this problem. The model is defined by the following assumptions, which we state with additional comments below. We also introduce definitions and notation used throughout this chapter; a table for the latter is also provided for ease of reference.

1. The target platform T comprises only processing resources of two types, which we refer to, without loss of generality, as CPUs and GPUs.

We assume there are r CPUs and s GPUs, and $q := r + s$ processing resources in total. When necessary, we denote a specific processor by p_a, where $a \in [1, q]$ is an index such that if $a \leq r$ then p_a is a CPU and if $a > r$ it is a GPU. Note that the labels should not always be interpreted literally. In particular, in numerical examples presented later, CPU cores are considered individually but entire GPUs regarded as discrete so that, for example, a platform comprising 4 GPUs and 4 octacore CPUs would be viewed as 4 GPU resources and $4 \times 8 = 32$ CPU resources. This is in keeping with much of the related literature and reasonable based on current programming practices [15], [127]. Similarly, a GPU could actually be any other accelerator, such as an FPGA or even a different CPU type.

2. The task graph G has been given and the scheduler has full access to its topology.

This is a standard assumption in offline scheduling. As noted in the previous section, for many runtime systems this is not currently possible, however for others it is, at least in some fashion. Of course, it should be recognized that there are applications whose corresponding task graph cannot be anticipated before runtime because of, for example, conditional statements. But in other cases this is an entirely reasonable assumption. For example, in many tiled numerical linear algebra applications, such as the Cholesky factorization example from Section 1.1, the entire DAG structure can be generated easily once a tile size has been specified. Given that these are widely-used across scientific computing, this cannot be viewed as a niche case. Note that by assuming the DAG is given we do not consider the important related problem of how to construct amenable task graphs for a given application.

3. All tasks are atomic and cannot be divided across multiple resources or aggregated to form larger tasks.

4. All processing resources can only execute a single task at any one time, which they do without preemption.

These tie into the previous assumption to some extent: ultimately, we are assuming that the task graph has been formed in such a way that these assumptions are sensible. Given current accelerated architectures, in the future it may be interesting to consider the possibility of individual tasks being processed by multiple CPU resources, as in [25], but we do not assume that is possible here. *Preemption* refers to pausing the execution of a task in order to execute another. However, we assume that once a processor begins to execute a task it must complete it without interruption.

5. All processors can in principle execute all tasks, albeit with different execution times.

Of course, there are plenty of examples one can imagine in which this isn't true—but likewise many in which this is the case. For example, in NLA, implementations of the BLAS are widely-available for both CPU and GPU.

6. Each task has a single scalar computation time on each processor type.

The execution time of task t_i on any CPU is denoted by c_i and on any GPU by g_i. Where necessary, we denote the processing time of task t_i on the specific processor p_a by $W_i^a \in \{c_i, g_i\}$. In reality, the real execution time of any task on any processor will never be known precisely beforehand. However, modern runtime systems generally use good performance models that allow this estimation to be made based on previous execution traces or other similar data [127]. Our own small-scale experimentation with NLA kernels, described in Section 2.5.1, also suggests that for certain common task types execution times tend to be tight.

7. Task execution times on CPU and GPU are *unrelated*, in the sense that the *acceleration ratio* $a_i := c_i / g_i$ may differ for distinct tasks.

This is one of the key features of CPU/GPU scheduling and has often been observed empirically: for example, in NLA, GPUs are typically much faster than CPUs at multiplying or factorizing very large matrices—classic *embarrassingly parallel* [91, p. 182] operations—but the converse may be true when performing smaller, more serial tasks [5], [141]. Note in particular that the acceleration ratio of a task may actually be less than one if its execution time is smaller on CPU than GPU. Unrelated processors are known to make scheduling problems more difficult, as described in Section 2.2.2.

8. *Communication delays* can occur.

In much of the CPU/GPU scheduling literature it is assumed that communication times—including all latency and data movement—are either negligible relative to the task processing times or sufficiently small that they can be safely ignored during scheduling. In some application areas this is reasonable, at least in theory, such as NLA when tile sizes are chosen to be sufficiently large [21], [121]. However, in practice, and for many other applications, this is not always the case. Therefore in this chapter we consider nonzero communication delays.

9. The platform is fully connected and all processor latencies and interconnect bandwidths are constant throughout.

By assuming that bandwidths are the same for all time we are neglecting contention for the communication resources. This corresponds to the classic *macro-dataflow* model [140], which is widely-used in the scheduling literature but somewhat unrealistic, perhaps especially for multicore architectures [4]. It should however also be noted that the kind of heuristics that we consider here can typically be extended in a straightforward manner for alternative models; for example, see [17] for an adaptation of HEFT to a one-port communication model.

10. The communication delay between any two tasks scheduled on the same processor, or on any pair of CPUs, is zero.

Disregarding communication delays between tasks on the same processor is a common assumption in the literature and reflects the fact that memory accesses tend to be much faster than data movement [17]. Similarly, we assume that all CPU-CPU communications are zero to reflect the likelihood that CPU resources are individual cores in a shared memory architecture. Of course, these assumptions are not representative of all possible architectures but are intended to be broadly reasonable for an accelerated platform today.

11. For every pair of communicating tasks (t_i, t_k) there is a single possible nonzero communication delay d_{ik}. In other words, we assume that the CPU-GPU, GPU-CPU and GPU-GPU communication delays are identical.

Traditionally, the communication delay W_{ik}^{ab} between t_i and t_k when they are scheduled on processors p_a and p_b, respectively, is modeled by

$$W_{ik}^{ab} := L_{ab} + \frac{D_{ik}}{B_{ab}}, \tag{2.1}$$

where L_{ab} is the relevant latency, B_{ab} is the bandwidth of the link between the processors, and D_{ik} is the amount of data that must be moved between the tasks [131]. We assume that the data movement cost dominates the latency and bandwidths are both symmetric ($B_{ab} = B_{ba}$) and similar for all of the links, so that there are only two possible values that W_{ik}^{ab} may take: zero, when $a = b$ or both p_a and p_b are CPUs, and some nonzero scalar d_{ik} in all other cases. In theory, modern inter-GPU interconnects are often faster than the links between the different processor types, so we could reasonably argue that the communication delays will be smaller in that case, but in practice realizing these

Table 2.1: Notation specific to this chapter.

r	Number of CPU resources
s	Number of GPU resources
q	Total number of processing resources, $q = r + s$
p_a	An individual processing resource, indexed by $a = 1, \dots, q$
c_i	CPU computation time of task t_i
g_i	GPU computation time of task t_i
a_i	Acceleration ratio of task t_i, $a_i = c_i / g_i$
W_i^a	Computation time of task t_i on processor p_a, $W_i^a \in \{c_i, g_i\}$
d_{ik}	Nonzero communication delay between tasks t_i and t_k
W_{ik}^{ab}	Communication delay between (t_i, t_k) on (p_a, p_b), $W_{ik}^{ab} \in \{0, d_{ik}\}$

theoretical performance peaks is often difficult [73], so we do not make that assumption. Questions can certainly be asked whether the communication model defined by Eq. (2.1) is sufficient for the complex memory architectures of modern accelerated platforms. However, most of the work presented here does not depend on the specifics of how the communication delays are estimated, only that it can be done somehow.

12. A *schedule* maps all tasks to the processors that should execute them and specifies the times at which their execution should begin.

In particular, the latter means that the order in which processors will execute their assigned tasks is also fixed.

13. A schedule is *optimal* if it minimizes the makespan over the set of all possible schedules.

Minimizing the makespan—the application runtime—is the most common aim in practice so is the natural choice. Other objectives, such as reducing total energy expenditure, are not explicitly considered here, although we suspect that much of what follows may be applicable for alternative objectives if costs are suitably redefined. The problem of optimizing two or more objectives simultaneously is however definitely beyond our scope.

For ease of reference we provide Table 2.1, which summarizes the notation used throughout this chapter. (Recall that we also retain the notation introduced in Table 1.1.)

2.2.2 Complexity

It should be clear that the DAG scheduling problem for CPU and GPU is simply an instance of a more general combinatorial optimization problem in which we have a set of *jobs* with various processing times and a collection of *machines* that must process them in order to optimize some *objective function*. Graham et al. [58] introduced a compact notation for classifying such problems, which was later extended by by Veltman, Lageweg and Lenstra [135], using three fields $\alpha|\beta|\gamma$, where α describes the machines, β the jobs and γ the objective function. In this notation, our problem is described by $(Pr, Ps) \mid prec, com \mid C_{\max}$, where the first field indicates that we have r identical parallel processors of one type and s of another, the second that the tasks are precedence constrained with communication delays, and the third that our goal is to minimize the makespan.

Unfortunately, even simpler related problems are known to be intractable. For example, it is well-known that the homogeneous parallel scheduling problem $P \mid prec \mid C_{\max}$ is NP-hard [72], with rare exceptions such as when $P = 2$ and all costs are uniform [40]. Likewise, the $P \mid prec,\ c = 1 \mid C_{\max}$ problem with unit communication delays and task processing times is also NP-hard [101]. Our problem is at least as difficult as these and therefore also falls in that class. However, it is less difficult than the corresponding problem $R \mid prec,\ com \mid C_{\max}$ with arbitrary unrelated processors.

2.3 Heuristics and approximation algorithms

Approximation algorithms for the $(Pr, Ps) \mid prec,\ com \mid C_{\max}$ problem are rare. However, at least one constant-approximation algorithm has recently been established for (a variant of) the problem [8], [9], an extension of a previous algorithm for the corresponding problem without communication delays [67]. These algorithms are discussed in greater detail in Section 2.3.3. To the best of our knowledge, it is not known whether $(Pr, Ps) \mid prec,\ com \mid C_{\max}$ permits polynomial-time approximation schemes (PTAS). It has been shown that both $P \mid prec \mid C_{\max}$ and $R \mid prec \mid C_{\max}$ do [61] but in general there are few results concerning scheduling problems with communication delays.

One notable approximation algorithm for the $P \mid prec \mid C_{\max}$ problem is Graham's classic *List Scheduling* (LS) algorithm [57], which proceeds as follows. First, a topologically sorted list of the tasks is computed. Then, whenever a processor becomes idle it scans the list from the top until it finds the first task ready for scheduling, which it begins to execute. For m identical processors, LS is a $(2 - 1/m)$-approximation algorithm, which was proven (using a variant of the unique games conjecture proposed in [16]) to be the best possible by Svensson [124]. Unfortunately, LS can perform arbitrarily badly for heterogeneous processors: because processors are never idle when there are tasks available, a processor may execute a task for which it has an arbitrarily long processing time because all better-suited processors were busy when it became idle.

Given the scarcity of practical approximation algorithms in this area, in practice heuristics with good average-case performance are usually preferred. As stated at the beginning of this chapter, most of those used for scheduling accelerated platforms were originally designed for the $R \mid prec, com \mid C_{\max}$ problem, so we give a brief overview here. Broadly speaking, they can be divided into four categories, with occasional overlap: *guided-random search*, *clustering*, *duplication-based*, and *listing* heuristics [131]. The first is a term used for any method that generates a large population of potential schedules and then selects the best among them. Typically these are more general optimization schemes such as genetic algorithms which are refined for the task scheduling problem. As a rule, such methods tend to find high-quality schedules but take a long time to do so and are therefore often impractical; for example, a comparative study by Braun et al. found that genetic algorithms usually obtained superior schedules to all other alternatives when scheduling sets of independent tasks but could take up to 300 times as long [30].

Clustering heuristics work by first grouping all tasks in the DAG and then scheduling each cluster to a single processing resource, with the aim of reducing communication costs [56]. This can be useful when such costs predominate, but the downside tends to be that the initial clustering step is expensive [131]. Duplication-based heuristics also attempt to reduce communication costs, this time by ensuring that communicating tasks are scheduled on the same resource even if this requires them to be replicated—i.e., the same task may be redundantly scheduled in more than one place [7]. Although they can perform well, they also tend to have high time-complexity bounds since controlling the amount of duplication is tricky: too much can lead to the system becoming clogged, with duplicated tasks obstructing the optimal scheduling of others. In addition, it may not always be possible in practice to duplicate tasks in the required manner.

Listing heuristics have a simple two-step structure: a list of all tasks is constructed in the *task prioritization* phase and they are then scheduled in this order according to some rule in the *processor selection* phase [131]. This framework is sometimes known as *Ordered List Scheduling* (OLS) to distinguish from the LS algorithm as described above [111]. Listing heuristics are the most popular type in practice since they are typically competitive with the alternatives whilst also being cheaper. The most prominent listing heuristic in this area is HEFT, which combines low time-complexity and good schedule quality; Canon et al. [35] compared twenty DAG scheduling heuristics empirically and found that HEFT's schedules were almost always among the shortest.

2.3.1 HEFT

To describe HEFT, first we must introduce the concept of the *critical path*. The term comes from project management, where it is defined as the longest sequence of activities that must be done in order to complete a project [68]. In a scheduling context, the critical path is analogously defined as the longest weighted path through the task graph. Note that this can only be calculated once a schedule has been computed and the DAG weights are fixed. The length of the critical path from source to sink clearly gives a lower bound on the schedule makespan. Moreover, the critical path length between *any* task and the sink gives a lower bound on the future schedule costs after that task has been processed. For *homogeneous* processors, the critical path can generally be anticipated before scheduling because all tasks have only a single possible execution time. With this in mind, a natural approach for a listing heuristic is to prioritize all tasks according to the length of their critical paths to the sink, the idea being that tasks with the greatest downward path length contribute most towards the eventual makespan and should therefore be given the highest priority. This has proven effective in practice for various homogeneous scheduling problems [1], [40], [70].

Unfortunately, it isn't obvious how the critical path can be computed before scheduling for *heterogeneous* processors, since there are multiple values each of the DAG weights may take and therefore typically many paths that may become critical. Extending the older *Modified Critical Path* heuristic [140], HEFT's solution is to set all DAG weights to their *mean* values. We define the mean computation time $\overline{w_i}$ of task t_i by

$$\overline{w_i} := \frac{1}{q}\sum_{a=1}^{q} W_i^a = \frac{rc_i + sg_i}{q}, \tag{2.2}$$

where the middle expression is how the mean would be calculated for generic heterogeneous processors and the expression on the right is its simplification under the accelerated scheduling model that we consider in this chapter. Similarly, the mean communication delay between t_i and t_k is defined through

$$\overline{w_{ik}} := \frac{1}{q^2}\sum_{a=1}^{q}\sum_{b=1}^{q} W_{ik}^{ab} = \frac{s(2r+s-1)d_{ik}}{q^2}. \tag{2.3}$$

After the DAG weights are set to their mean values, critical path lengths are computed in the usual way—i.e., through dynamic programming. Starting from the exit task, we set its *upward rank* (sometimes also called the *bottom level*) $u_n = \overline{w_n}$, then move up the DAG and recursively compute

$$u_i = \overline{w_i} + \max_{k\in\Gamma_i^+}(\overline{w_{ik}} + u_k) \tag{2.4}$$

for all other tasks. The task prioritization phase then concludes by listing all tasks in decreasing order of upward rank, with ties broken arbitrarily.

The processor selection phase of HEFT is straightforward: we move down the list and schedule each task on the processor expected to finish it at the earliest time. However, HEFT follows an *insertion-based* policy that allows tasks to be inserted in a processor's load between two others that have already been scheduled, assuming precedence constraints are still respected. For a generic task t_i, let S_i^a denote its start time when it is scheduled on processor p_a and F_i^a denote the corresponding finish time. Clearly we have

$$F_i^a = S_i^a + W_i^a \tag{2.5}$$

but the question remains of how we compute S_i^a in the first place. There are two cases to consider. If t_i is the entry task (i.e., $i = 1$), then trivially we have $S_1^a = 0$. If t_i is not the entry task, then p_a cannot begin to execute t_i until all of its parents have been processed and any necessary communication delays have expired. In other words, the earliest possible time that p_a could possibly begin to execute t_i, disregarding any other tasks that may already be scheduled on it, is given by

$$D_i^a = \max_{h\in\Gamma_i^-}(F_h^b + W_{hi}^{ba}). \tag{2.6}$$

To identify when p_a can actually process t_i, we then scan the list of tasks already scheduled on p_a in order of their expected start time and set S_i^a to the earliest time such that $S_i^a \geq D_i^a$ and there is no other task scheduled to start in the time interval $[S_i^a, S_i^a + W_i^a]$.

A complete description of HEFT is given in Algorithm 2.1. HEFT has time complexity $O(v \cdot q)$ [131]. For dense DAGs, the number of edges v is proportional to n^2, so the complexity is effectively $O(n^2 q)$. This compares well with many of the alternatives: with communication delays, quadratic complexity in n is about the best that we can realistically expect since each edge must be inspected at least once.

(Note that, rather than sorting all of the tasks into a list and scheduling tasks in that order, we could instead maintain a list of those tasks that are currently ready for scheduling, initialized with the source, and repeatedly select the one with the highest rank, updating the list as new tasks become ready. Since upward ranking defines a topological sort of the tasks, this formulation is equivalent to that given above, but is sometimes preferred.)

Algorithm 2.1: HEFT.

```
1  Set the mean weight of all tasks using Eq. (2.2)
2  Set the mean weight of all edges using Eq. (2.3)
3  Compute $u_i$ for all tasks according to Eq. (2.4)
4  Sort the tasks into a priority list $L$ by non-increasing order of $u_i$
5  for $t \in L$ do
6      for $a = 1, \dots, q$ do
7          Compute $F_i^a$ using Eqns. (2.5) and (2.6)
8      end
9      $p_{\min} := \arg\min_a F_i^a$
10     Schedule $t_i$ on $p_{\min}$
11 end
```

HEFT has a long record of good performance in practice, including for CPU/GPU platforms, and has arguably become the benchmark against which all new heterogeneous scheduling heuristics are measured [19]. However, despite this empirical excellence, it has no performance guarantee. Bleuse et al. [26] showed that even for the (Pr, Ps) problem without communication delays and a single GPU (i.e., $s = 1$), HEFT's approximation ratio is at least $r/2$. Similarly, Amaris et al. [10] proved that if $s \leq \sqrt{r}$ then the approximation ratio is greater than $\frac{r+s}{s^2}(1 - e^{-s})$, even for tasks without precedence constraints.

CPOP. Given the versatility of HEFT, many extensions and similar heuristics have been proposed. Indeed, introduced by the original authors in the same paper was *Critical-Path-On-a-Processor* (CPOP), which largely proceeds as in HEFT, except for tasks that are identified as being on the critical path. As the name suggests, these are always scheduled on the same processor, in an attempt to reduce communication costs between them. To identify critical path tasks, both upward and downward ranks are computed, where the latter are defined by setting $d_1 = 0$ for the source t_1, then moving down the DAG and recursively computing

$$d_i = \max_{h \in \Gamma_i^-}(\overline{w_{hi}} + d_h) \tag{2.7}$$

for all other tasks. Intuitively, the sum of the upward and downward ranks of a task is the longest path from source to sink which passes through that task, so that all tasks with the greatest such sum are expected to be critical. Alternative ways of estimating the critical path are possible in the CPOP framework; Gregg and Vasudevan propose one such example [134]. Empirically, CPOP has usually proven inferior to HEFT for general DAGs and heterogeneous platforms, although it can be superior when communication costs are high [35], [131].

Alternative rankings. HEFT uses mean values to set task and edge weights but other averages could just as easily be used instead. Zhao and Sakellariou [147] considered several choices, including the median, maximum and minimum, with both upward and downward ranking, through extensive numerical experimentation. Ultimately they concluded that mean values certainly did not appear to be an obviously superior choice to the others, although none of the alternatives dominated the comparison either. Perhaps the bigger takeaway was that, although significant runtime reductions are possible by following the best task priority list for a given DAG, actually identifying which method will produce that priority list is very difficult. More decisively, their experiments suggested that upward ranking almost always outperformed downward, although again there were instances in which this was not the case.

Lookahead. Once all task priorities have been computed, the processor selection phase of HEFT seems obvious: the very word *priority* suggests that the natural approach is to act greedily. Ultimately, the question is, when do we *not* schedule a task on the resource which is expected to finish it at the earliest time? The obvious answer is: when we expect that doing so will lead to a longer schedule in the end. HEFT with Lookahead [23] attempts to infer the future effects of processor selections by simulating the scheduling of a task's children (in the usual greedy way), assuming that they immediately become ready—i.e., ignoring any other unscheduled parents they may have. The processor which minimizes some average of the expected child finish times is then selected; the maximum and a type of weighted mean were considered experimentally in the original paper, with the latter performing marginally better. Note that the lookahead horizon can be extended arbitrarily by assuming that the children of the initial child tasks are also scheduled in the same manner, although this is subject to diminishing returns since the error from disregarding unscheduled parents begins to accumulate. HEFT with Lookahead is reported to perform slightly but consistently better than the standard algorithm, with the downside of increasing the time complexity to $O(n^4)$.

HEFT-NC. All of the HEFT variants discussed above were intended for generic heterogeneous environments. However, HEFT-NC (*No Cross*) from Shetti, Fahmy and Bretschneider [114] was designed for accelerated platforms in particular. The algorithm has the same basic structure as HEFT but makes modifications to both the task prioritization and processor selection phases. For the former, rather than using mean task computation costs to set task weights $\overline{w_i}$ in the upward ranking scheme given by (2.4), they instead define

$$\overline{w_i} = \frac{\max(c_i, g_i) - \min(c_i, g_i)}{\max(c_i, g_i) / \min(c_i, g_i)}, \tag{2.8}$$

with average communication delays $\overline{w_{ik}}$ all set to zero. The motivation is that rather than prioritizing tasks according to their average size, tasks which have the strongest *preference* for one processor type should be prioritized instead. Indeed, equation (2.8) is an amalgamation of two more intuitive ways of computing a preference score based on the task weights, namely the difference (numerator) and ratio (denominator). It isn't clear to us, however, that the suggested combination of the two is successful in its aim. For example, consider a task t_1 such that $c_1 = 10$ and $g_1 = 1$, and another task t_2 with $c_2 = 11$ and $g_2 = 10$. Then using Eq. (2.8) we would define

$$\overline{w_1} = \frac{10-1}{10/1} = \frac{9}{10} \quad \text{and} \quad \overline{w_2} = \frac{11-10}{11/10} = \frac{10}{11},$$

so that $\overline{w_1} < \overline{w_2}$ and t_2 has the greatest weight, despite the fact that t_1 clearly has a stronger preference. Nonetheless, we will evaluate the utility of this method for computing task priorities experimentally in Section 2.5.3.

The processor selection phase of HEFT-NC proceeds largely as in HEFT, except for when the chosen processor p_a does not also minimize the task computation cost—i.e., $W_i^a \neq \min(c_i, g_i)$. In that case, the processor of the other type with the minimal expected finish time, call it p_b, is identified and a quantity

$$Y = \frac{F_i^b - F_i^a}{F_i^b / F_i^a}$$

is computed. Then, if $\overline{w_i}/Y \leq X$, where X is a parameter called the *cross threshold*, t_i is scheduled on p_a; otherwise, it is scheduled on p_b. Conceptually, the basic idea is to weigh the expected makespan gain from choosing the fastest processor against the strength of the task's preference for processors of the other type. The immediate issue is how we find a good cross threshold; $X = 0.3$ is suggested based on numerical experimentation but finding the optimal choice in general is an open question. Shetti, Fahmy and Bretschneider compared their new algorithm to HEFT for 2000 randomly generated DAGs, concluding that it usually achieved a smaller makespan, with an average reduction of roughly ten percent.

2.3.2 PEFT

Predict Earliest Finish Time (PEFT) [12] was the first heuristic to incorporate a degree of lookahead into the HEFT framework without increasing the exponent of n in the time complexity. In the processor selection phase, finish time estimates for the current task on each of the processors F_i^a are added to *optimistic* estimates O_i^a of the future costs we expect to incur given that selection, and the processor which optimizes the sum of the two is chosen—i.e., we schedule task t_i on the processor p_{opt} defined by

$$p_{\text{opt}} := \min_a \left(F_i^a + O_i^a\right). \tag{2.9}$$

The optimistic cost estimates O_i^a are computed before the scheduling begins in the following manner. Starting from the sink, we set $O_n^a = 0$ for all $a = 1, \ldots, q$, then move upward and recursively compute

$$O_i^a = \max_{k \in \Gamma_i^+} \Big(\min_{b=1,\ldots,q} \left(\delta_{ab} \overline{w_{ik}} + W_k^b + O_k^b \right) \Big), \quad \forall a = 1, \ldots, q, \tag{2.10}$$

for all other i, where $\delta_{ab} = 1$, if $a = b$, and 0 otherwise.

Note that O_i^a is intended to be a lower bound on the future schedule costs, assuming that task t_i is scheduled on processor p_a; it is called *optimistic* because all processor contention is ignored. With this in mind, it isn't clear to us why the average value $\overline{w_{ik}}$ is used in equation (2.10), given that within the minimization each parent task is already assumed to be scheduled on processor p_b—why not use the actual communication delay W_{ik}^{ab} instead? Indeed, defining O_i^a using average values in some sense contradicts the word *optimistic* since it does not in fact give a true lower bound on the remaining schedule costs; see Chapter 3 for more on this. Therefore we suggest that O_i^a should instead be defined by

$$O_i^a = \max_{k \in \Gamma_i^+} \left(\min_b \left(W_{ik}^{ab} + W_k^b + O_k^b \right) \right). \tag{2.11}$$

At any rate, under the assumptions of our communication model there would appear to be little difference between the two definitions, but in Section 2.5.4 we investigate which seems to perform best empirically.

Computing the O_i^a is an $O(qn^2)$ operation—i.e., the same complexity in n as HEFT. Furthermore, note that in the case of multiple identical processors, the cost is significantly reduced since the optimistic cost estimates depend only on the processor type. In particular, under our accelerated model, for each task t_i we only have two values O_i^c and O_i^g, representing the estimates assuming that the task is scheduled on a CPU or GPU, respectively. Moreover, these can be computed recursively by setting $O_n^c = O_n^g = 0$ then working up the DAG using

$$O_i^c = \max_{k \in \Gamma_i^+} \left(\min \left(d_{ik} + g_k + O_k^g,\ c_k + O_k^c \right) \right) \tag{2.12}$$

and

$$O_i^g = \max_{k \in \Gamma_i^+} \left(\min \left(d_{ik} + c_k + O_k^c,\ g_k + O_k^g \right) \right), \tag{2.13}$$

to calculate the optimistic costs for a generic task t_i. (These formulae follow the variant defined by Eq. (2.11); under (2.10) the d_{ik} would be replaced by the average cost $\overline{w_{ik}}$ instead.)

Given the computational effort expended in computing the optimistic costs—and the fact that they are very similar in nature to the upward ranks u_i—it seems sensible to use them for computing task priorities as well. Hence PEFT defines task priorities O_i through

$$O_i = \frac{1}{q}\sum_a O_i^a = \frac{1}{q}(mO_i^c + kO_i^g) \tag{2.14}$$

where the second expression is how the ranks would be calculated under our accelerated scheduling model. There are two differences with HEFT's upward ranking step worth noting. First, the task ranks do not include the cost of the task itself (although it is suggested that the savings made through the lookahead are more beneficial overall). Second, the O_i do not induce a topological sort of the tasks—i.e., $k \in \Gamma_i^+ \not\Longrightarrow O_i \geq O_k$. The algorithm therefore proceeds by selecting the task with the largest rank from the current set of ready tasks for scheduling, rather than following an ordered list.

PEFT was reported to obtain shorter schedules than HEFT on average for a large collection of real and randomly generated DAGs. Performance was particularly good when the number of processors is high, which is perhaps to be expected given that ignoring contention when computing the optimistic costs is more reasonable in that case. However, it isn't clear that this trend will hold when there are lots of one kind of processor (i.e., CPUs) but relatively few of another (GPUs).

2.3.3 HLP

In 2015, Kedad-Sidhoum, Monna and Trystam [67] proposed the first approximation algorithm for the $(Pr, Ps) \mid prec \mid C_{\max}$ problem, which defines a framework that we will refer to as *Heterogeneous Linear Programming* (HLP). The basic idea is to divide the scheduling problem into two stages: first, an *assignment* to a processor type is computed for all tasks, then tasks are scheduled on a specific processor according to their assignment using a good approximation algorithm for the $P \mid prec \mid C_{\max}$ problem (i.e., Graham's LS algorithm).

Let x_i be a binary assignment variable such that $x_i = 1$ if task t_i is assigned to CPUs, and 0 otherwise, and let $\alpha = \{x_1, \ldots, x_n\}$ be a complete assignment of all the tasks. Clearly, we want to compute an assignment which gives us the most scope to minimize the makespan of the corresponding schedule; we will refer to this the assignment problem. There are two classic lower bounds on the makespan of any schedule: the critical path of the task graph, and the total amount of work done, divided by the number of processors—sometimes referred to as the *area bound*. If we neglect communication delays (as in the original paper), the critical path L_α for a given assignment α is straightforward to compute since all tasks weights are known and edge weights are all zero. Likewise, the area bound is simply the maximum of the work assigned to the CPUs and GPUs. Let C_α be the total amount of work done on CPUs and G_α be the corresponding GPU workload under the assignment α, so that

$$C_\alpha = \sum_{i=1}^{n} c_i x_i \quad \text{and} \quad G_\alpha = \sum_{i=1}^{n} g_i(1 - x_i).$$

Then for the makespan $C_{\max}$ we must have $C_{\max} \geq \lambda_\alpha := \max\{L_\alpha, \frac{C_\alpha}{r}, \frac{G_\alpha}{s}\}$ for all α, so that the assignment problem is to find α such that λ_α is minimized. This can be expressed as the following linear program (LP):

Minimize λ such that

$$\begin{aligned}
&F_i + c_i x_i + g_i(1 - x_i) \leq F_k, \quad \forall k \in \Gamma_i^+, \forall i, \\
&0 \leq F_i \leq \lambda, \quad \forall i, \\
&\sum_{i=1}^{n} c_i x_i \leq r\lambda, \\
&\sum_{i=1}^{n} g_i(1 - x_i) \leq s\lambda, \\
&x_i \in \{0, 1\}, \quad \forall i.
\end{aligned}$$

Unfortunately, the LP above cannot be solved in polynomial time. However, if we relax the integrality constraints on the x_i—i.e., suppose $x_i \in [0, 1]$—then it can be. Let $\alpha' = (x_1', \ldots, x_n')$ be the solution of the relaxed LP and let $\alpha^* = (x_1^*, \ldots, x_n^*)$ be the feasible assignment computed by rounding the corresponding x_i' values to the nearest integer (i.e., 0 or 1). HLP is defined by following the assignment α^* and employing Graham's LS algorithm to schedule tasks on the specific resource of their assigned type.

The general approach of defining a problem as an LP, relaxing integrality constraints in order to solve it polynomially and then rounding the answer to get a feasible solution is an example of a common technique called *randomized rounding* [100], which is often used to design approximation algorithms for NP-hard problems. And so it proved here: Kedad-Sidhoum, Monna and Trystam showed that HLP achieves a constant approximation ratio of 6 and was therefore the first algorithm with a performance guarantee for the $(Pr, Ps) \mid prec \mid C_{\max}$ problem.

Despite this impressive result, in practice we are often more concerned with average performance than a worst-case bound and HLP does not appear to improve on, for example, HEFT in that regard [20]. With this in mind, Amaris et al. [10] proposed a refinement of the algorithm called HLP-OLS. The assignment LP is formulated, relaxed and solved as before, but instead of using the LS algorithm to schedule the tasks, Ordered List Scheduling (OLS) is used instead, as in HEFT. To construct the list we compute upward ranks, but rather than using mean values for the task weights, we use the weight indicated by the assignment (i.e., if $x_i^* = 1$, we take the weight of t_i to be c_i, and so on). The tasks are then sorted into a list according to their rank, and scheduled in this order on the processor of their assigned type that is expected to complete them at the earliest time. HLP-OLS is reported to achieve better empirical performance than the original algorithm. Moreover, it has the same approximation ratio (i.e., 6). Amaris et al. also generalized HLP for $Q \geq 2$ different processor types and proved that it achieves an approximation ratio of $Q(Q+1)$. However, their simulation results suggest that average performance is considerably poorer than HEFT for $Q = 3$.

All of the aforementioned versions of HLP disregard communication, but Aba, Zaourar and Munier [8], [9] extended the framework to incorporate communication delays. Their model is broadly similar to our own, although simpler in that delays only ever occur between different processor types (i.e., communication delays between distinct GPUs are assumed to be negligible). However, even this proves somewhat problematic to integrate into HLP whilst retaining the approximation guarantee. Two variants were proposed: a non-polynomial time 6-approximation and, later, a polynomial time 6τ-approximation, for some instance-specific τ. In both cases, formulating the assignment LPs requires some subtlety since more straightforward choices do not permit the performance guarantees, a clear drawback of this approach relative to heuristics like HEFT which handle alternative communication models more easily. The authors also extend the HLP algorithm for a simple energy model in which each task has a single CPU and GPU energy cost and the sum of such costs must meet some budget. This is done by adding the budget constraint to the LP before solving for the assignment.

Aside from the issues already mentioned, the main problem with the HLP approach is that solving the assignment LP tends to be expensive, even when it can be done polynomially; using IBM's popular `CPLEX` solver, runtimes an order of magnitude higher than for

HEFT are reported [8], [9]. Moreover, it is isn't clear that the gains are worth the extra computational effort on average, although a modified version of HLP incorporating *spoliation* (see below) was found to be the best of all the algorithms considered in [19].

2.3.4 HeteroPrio

Whereas HLP was created with theoretical guarantees in mind, another algorithm proposed in recent years has a very different origin, first being developed as a practical method for a single application [6] and then being extended to the general case because of its good performance. Like HLP, there are several variants of *HeteroPrio* [2], [18], [20] but the motivation is the same: rather than asking which processor a given task should be scheduled on, as in HEFT and similar heuristics, why not change perspective and ask which of the tasks a given processor should select instead?

HeteroPrio simulates the execution of the DAG offline and repeats the following procedure until all tasks have been processed. Whenever a processor p_a becomes idle, it considers the set of currently ready tasks R. These tasks are assumed to be sorted in ascending order of their acceleration ratio $a_i = c_i / g_i$, with ties broken according to some prioritization scheme (e.g., upward ranks as in HEFT). If R is not empty, then p_a pops a task from the *head* of the ready task list if it is a CPU, or from the *tail* if it is a GPU. If R is empty, then p_a considers the set of tasks which are currently being executed by other processors, in order of their priority. If any task is expected to finish earlier if it were processed by p_a, then it is *spoliated*: p_a steals the task and begins its execution again from scratch. Note that this is distinct from preemption because all work done by the other processor is thrown away; in the offline case that we consider here, the task's execution on the other processor is never even begun and spoliation simply means changing an earlier scheduling decision (something which is not possible in, for example, HEFT).

The motivation for the different order tasks are selected by CPUs and GPUs comes from the fact that HeteroPrio was originally designed for scheduling independent tasks and only later extended to DAGs by operating on the ready task set. For a set of independent tasks, it can be shown that the optimal solution to the relaxed assignment problem—i.e., assuming $x_i \in [0, 1]$—can be obtained through a simple greedy algorithm that sorts all tasks in ascending order of their acceleration ratio then, at the same rate, assigns tasks at the head to the CPUs and tasks at the bottom to the GPUs, with the final *pivot* task being split between the two types such that the loads are balanced, if necessary [18], [19].

HeteroPrio has a good performance guarantee for independent tasks, achieving a competitive ratio of $2 + \sqrt{2} \approx 3.41$ in the general case, which reduces to 2 if the optimal schedule makespan is greater than $\max\{c_i, g_i\}$ for all $i = 1, \ldots, n$ [18]. This compares well with the best cheap approximation algorithms in that case. The bound is less impressive for task DAGs, with a variant of the basic algorithm shown to achieve an approximation ratio of $r + s$ for the problem without communication delays [18]. Interestingly, although one might assume it would complicate the analysis, spoliation is key to the proof of this bound.

In the average case, HeteroPrio has been found to perform very well, both in terms of runtime and schedule quality; comparisons with other algorithms such as HEFT and HLP in the literature have typically found that it is competitive, at least without communication delays [2], [19], [20]. Furthermore, a modification of HeteroPrio which takes data locality into account—and therefore communication—has been proposed [29]. Extending HeteroPrio to heterogeneous platforms with arbitrarily many different processor types was considered by Kumar in his thesis [71], with a focus on designing sensible counterparts to the acceleration ratio in that case. Performance was reported to be good, although again communication delays were not considered.

2.4 PRIORITY-BASED HEURISTICS

The recent survey paper from Beaumont et al. [19] compared HEFT, HeteroPrio and HLP (among others) experimentally using the `pmtool` [53] simulation environment in StarPU. Despite the fact that the other two algorithms were specifically designed for accelerated platforms, HEFT was still competitive. Furthermore, communication delays were not considered in that study; one of the advantages of HEFT, certainly relative to HLP, is that communication is straightforward to incorporate. Moreover, although the standard HEFT algorithm was used as a reference there—as it tends to be elsewhere in the accelerated scheduling literature—extensions such as HEFT with Lookahead or similar heuristics such as PEFT were not included. With these points in mind, we investigated how the general framework underlying *priority-based* heuristics such as HEFT can be optimized for accelerated architectures.

We define a generic priority-based heuristic as repeating the following procedure until all tasks have been scheduled:

1. The task with the highest priority is selected from the set of ready tasks (task prioritization);
2. The task is scheduled on a processor according to some rule (processor selection);
3. The set of ready tasks is updated.

This definition is very similar to a classic listing heuristic, with the exception that we do not assume priorities define a partial order of the tasks so we cannot (necessarily) sort them into a list and schedule them in that order. If the priorities *do* induce a valid topological sort then the two definitions are equivalent in the offline case that we study here; however, we choose to formulate the framework in this way because it is both more flexible and more easily adapted to the online scheduling models employed by most runtime systems. In this section we propose multiple alternatives for the task prioritization and processor selection phases of this simple algorithm.

2.4.1 Task prioritization

Ultimately, comparing the priorities of two ready tasks should tell us how important it is that one is scheduled before the other. Clearly, there are many different ways that task priorities can be defined, but upward ranking as defined in HEFT has proven effective. However, it isn't clear that using mean values is the best choice of average, even for general heterogeneous platforms; for accelerated platforms with only two possible task weights—which may be very different—this is especially true. As mentioned in Section 2.3.1, Zhao and Sakellariou [147] studied this problem (for generic heterogeneous platforms), comparing the following averaging schemes, defined by the average types they use to set the graph weights when computing upward ranks:

1. *Mean* (M), all weights are set to their mean values (the default);
2. *Median* (MD), all weights are set to their median values;

3. *Worst* (W), all task weights are set with their worst (i.e., largest) possible values and edge weights are set to the worst possible value given the assignment indicated by the weights of the two tasks they connect;

4. *Simple worst* (SW), all weights are set to their worst possible values;

5. *Best* (B), all task weights are set with their best (i.e., smallest) possible values and edge weights are set to the best possible value given the assignment indicated by the weights of the two tasks they connect;

6. *Simple best* (SB), all task weights are set to their best possible values.

They found that the M ranking was not superior to all of the others, although none of the alternatives was clearly dominant either. With this in mind, it seems reasonable to repeat their comparison for accelerated platforms in particular. Indeed, it was observed in [20] that using minimum values for task weights in HEFT —i.e., the B/SB rankings—led to better performance on accelerated platforms without communication delays; this was attributed to the fact that GPU processing times were always smaller and most tasks were eventually assigned to the GPUs, so that using GPU times as weights more accurately reflected the actual cost of the tasks.

In addition to the six averaging schemes listed above, below we describe eight other averages that may be used, giving a total of 14 different schemes. Each of these defines a complete task prioritization phase when upward ranks are computed using Eq. (2.4)—i.e., we compute u_i for all $i = 1, \dots, n$ through the averaging scheme and take the u_i to be task priorities. For ease of reference, we provide Table 2.2, which defines how each averaging scheme computes the weights $\overline{w_i}$ and $\overline{w_{ik}}$ of a generic task t_i and edge (t_i, t_k), respectively.

Alternative means. Consider what the upward ranks actually represent: estimates of the critical path length from tasks to the sink. Since downward tasks have not yet been scheduled, their weights cannot be anticipated without restricting the processor selection phase; by using average values over all processors, HEFT is effectively assuming that all processors are equally likely to be selected. But this seems unnatural: if a task's GPU execution time is much smaller than its CPU execution time, it is presumably more likely to be scheduled on the former. Therefore it seems reasonable to weight the mean according to the relative size of the processing times, with smaller values given more importance.

Table 2.2: Averaging schemes defined by how they compute the weight of a generic task t_i and edge $(t_i,\, t_k)$.

Name	$\overline{w_i}$	$\overline{w_{ik}}$
M	(2.2)	(2.3)
MD	Median of $\{W_i^a\}_{a=1,\ldots,q}$	Median of $\{W_{ik}^{ab}\}_{a,b=1,\ldots,q}$
B	$\min(c_i, g_i)$	0, if $\overline{w_i}$ and $\overline{w_k}$ same type, else d_{ik}
SB	$\min(c_i, g_i)$	0
W	$\max(c_i, g_i)$	0, if $\overline{w_i}$ and $\overline{w_k}$ both CPU, else d_{ik}
SW	$\max(c_i, g_i)$	d_{ik}
HM	(2.17)	(2.18)
SHM	(2.17)	0
GM	(2.19)	(2.20)
SGM	(2.19)	0
R	$\max(c_i, g_i) / \min(c_i, g_i)$	0
D	$\max(c_i, g_i) - \min(c_i, g_i)$	d_{ik}
NC	(2.8)	0
SD	Std. dev. of $\{W_i^a\}_{a=1,\ldots,q}$	Std. dev. of $\{W_{ik}^{ab}\}_{a,b=1,\ldots,q}$

In other words, we use the *harmonic* mean of the possible task weights, rather than the arithmetic one. Let

$$h_i = \sum_{a=1}^{q} \frac{1}{W_i^a} \quad \text{and} \quad m_i^a = \frac{1}{W_i^a h_i},\ \forall a = 1, \ldots q,$$

for all $i = 1, \ldots, n$. Then under this prioritization scheme the weight of task t_i would be

$$\overline{w_i} = \sum_{a=1}^{q} W_i^a m_i^a = \frac{q}{h_i}. \tag{2.15}$$

There is a minor issue when extending this idea to the edge weights, since zero is always one of the possible weights (from a processor to itself, or between any pair of CPUs) and therefore their harmonic mean is conventionally defined to be zero as well. We consider this possibility, however a more intuitive alternative may be to weight the values according to the probable weights of the communicating tasks. In particular, for a generic edge $(t_i,$ $t_k)$ define

$$m_{ik}^{ab} = m_i^a \cdot m_k^b = \frac{1}{W_i^a W_k^b h_i h_k} \quad \forall a, b = 1, \ldots, q,$$

and

$$\overline{w_{ik}} = \sum_{a=1}^{q} \sum_{b=1}^{q} W_{ik}^{ab} m_{ik}^{ab} = \frac{1}{h_i h_k} \sum_{a,b} \frac{W_{ik}^{ab}}{W_i^a W_k^b}. \tag{2.16}$$

Note that for the accelerated model that we consider here, the average task and edge weights $\overline{w_i}$ and $\overline{w_{ik}}$ simplify to

$$\overline{w_i} = \frac{q c_i g_i}{r g_i + s c_i} \tag{2.17}$$

and

$$\overline{w_{ik}} = \frac{d_{ik} r s(c_i g_k + c_k g_i) + d_{ik} c_i c_k s(s-1)}{(r g_i + s c_i)(r g_k + s c_k)}. \tag{2.18}$$

Following the lead of Zhao and Sakellariou [147], we refer to the averaging scheme defined by using Eq. (2.17) for the node weights and Eq. (2.18) for the edge weights as *harmonic mean* (HM), and the alternative defined by taking all edge weights to be zero as *simple harmonic mean* (SHM).

The other classical mean is the *geometric mean*, defined for a generic set of data points $(x_1, x_2, \ldots, x_n)$ as

$$\overline{x} = \left(\prod_{i=1}^{n} x_i \right)^{1/n}$$

and therefore for the weight of a task in our model as

$$\overline{w_i} = (c_i^r g_i^s)^{1/q}. \tag{2.19}$$

Although the intuitive justification is weaker than for the harmonic mean, the geometric mean is less sensitive to extreme values than the others so may be useful given the typical disparity between task CPU and GPU execution times. Like the harmonic mean, the geometric mean of all edge weights is zero, since that is always one of the possible values. However, a common workaround in such cases is to add one to all values, so that for our model the average weight of a generic edge (t_i, t_k) would be given by

$$\overline{w_{ik}} = (d_{ik} + 1)^{s(2r+s-1)/q}. \tag{2.20}$$

We refer to the averaging scheme defined by using a geometric mean for the task weights and Eq. (2.20) for the edge weights as *geometric mean* (GM), and the version with all edge weights taken to be zero as *simple geometric mean* (SGM).

Preference-based averages. As described in Section 2.3.1, equation (2.8) is used in the HEFT-NC [114] heuristic as an average which attempts to quantify how strongly tasks prefer one processor type relative to the other, rather than quantifying the relative sizes of the tasks. In particular, task weights are set using Eq. (2.8) before computing upward ranks, with all edge weights set to zero. We will refer to this averaging scheme as *no cross* (NC) from now on. Recall that Eq. (2.8) is a combination of two alternative methods for computing task preferences, namely the *difference* and *ratio* of the largest and smallest execution times, whose corresponding task prioritization schemes we will denote by D and R, respectively. Conceptually, when deciding which of two tasks to schedule first, rather than selecting the one expected to lie on a longer path through the remainder of the DAG—and therefore contribute most to future schedule costs—this approach aims to prioritize the task which lies on a path whose constituent tasks have the strongest preference for processors of one type. Leaving aside the issues with the NC weighting function that were identified previously, there are two other drawbacks with the three preference-based schemes from [114] that we can see.

1. It is difficult to incorporate communication delays, particularly in light of the fact that the smallest possible delay is always zero.

Since the minimum communication delay is always zero, NC and R in particular cannot be applied directly to the edges since we need to divide by the smallest value.

2. The number of processors of each type is not taken into consideration.

It seems somewhat counterintuitive that a task's preference is the same for platforms with a single GPU and either 1 or 100 CPUs.

An alternative average that may quantify task preferences and also account for the two issues highlighted above is the *standard deviation*. Intuitively, if the standard deviation of a task's possible processing times is small then the values are similar and the task therefore has weak preference for either type, whereas a large standard deviation suggests a stronger preference. Moreover, the standard deviation can easily handle zeroes and duplicated values so can be applied directly to the edges as well. Therefore, we will investigate how the corresponding averaging scheme—which we will refer to as SD—performs empirically compared with the NC, R and D schemes.

Autopsy. The reason average values are used when computing the upward rank is that scheduling decisions haven't been made yet for future tasks. But if task assignments were known then the majority of the weights would be determined and we can use those values, as in the HLP-OLS [10] algorithm. As discussed in Section 2.3.3, computing an optimal assignment using linear programming is expensive. However, a cheap alternative is to simply use the assignment derived from a schedule produced by any other heuristic, such as HEFT. In particular, we propose the following procedure, which we refer to as the *autopsy* method.

1. Get assignment α from the initial schedule.

2. Compute upward ranks using weights indicated by α and set as task priorities.

3. Schedule the graph again, but using new priorities and sticking to the assignment α.

Intuitively, the idea is to improve the schedule by re-running the heuristic used to compute it with more "accurate" task priorities. Since the assignment of the new schedule is identical to the old one, most of the lower bounds on the makespan—i.e., the path and area bounds as described in Section 2.3.3—remain the same so one would hope that a better task prioritization will reduce the gap. The only slightly awkward part here is the second step: under our model, most of the weights of the task graph are determined by the assignment α, with the exception of edges connecting two tasks scheduled on GPUs, which are zero if the tasks are on the same GPU, and the nonzero communication delay d_{ik} otherwise. In the spirit of HEFT, it seems most fitting to use the mean intra-GPU communication cost $\frac{s-1}{s}d_{ik}$ in that case. Of course, the autopsy method effectively doubles the computational effort, assuming a priority-based heuristic is used to compute the initial schedule, so it is imperative that the new schedule has a strong likelihood of improving on the old one; this will be evaluated experimentally later.

In this chapter we only consider scheduling with the aim of minimizing the makespan. In practice, although that is almost always one of the objectives, we often have others as well. In particular, reducing the energy expended during the application execution has become increasingly important in recent years. Commonly we have some sort of *budget* defining a limit on the total energy expenditure (or the financial cost associated with the energy expenditure) during the schedule execution; the aim is then to minimize the makespan whilst simultaneously meeting the budget constraint. One common approach for dealing with this kind of problem is to first compute an initial schedule which optimizes the objective (i.e., the makespan) and then modify this to meet the constraint in such a way that the degradation in the objective is minimized. Examples would be the *Dynamic Constraint Algorithm* (DCA) [99] or the LOSS variants proposed in [108]. Since energy costs will typically be defined by the assignment—i.e., whether a task is executed on a CPU or GPU, or data needs to be moved from CPU to GPU memory—the autopsy method can be combined with this technique, with an additional step between the first and second above in which the assignment α is modified in order to meet the budget. Although we do not consider this here, it is worth noting as direction of possible future research (see Section 2.6).

Optimistic costs. Intuitively, we can view upward ranking as determining task priorities by aggregating local—i.e., node and edge—comparisons of the graph weights. An alternative approach would be to instead determine task priorities by comparing quantities that represent more holistic estimates of "how good" it is to schedule the tasks on each processor. In particular, as described in Section 2.3.2, the PEFT [12] heuristic computes task priorities using the arithmetic mean of the set of *optimistic costs* over all processors. In the accelerated case, for a given task t_i, the latter consists of r copies of O_i^c and s copies of O_i^g. Clearly, any different average could be used instead, so it seems sensible to also consider the alternatives listed in Table 2.2. However, since we don't need to consider nodes and edges separately, it doesn't make sense to distinguish "simple" variants as was done there, so we do not define the SB, SW, SHM or SGM averaging schemes in this case. This gives us 10 possible averages over the sets of optimistic costs.

Apart from considering different average types, we make two minor changes to the task prioritization as defined in PEFT. First, we use the alternative optimistic costs defined by Eq. (2.11) rather than (2.10) because they give true lower bounds on the future schedule costs in the different cases. Second, rather than averaging optimistic costs O_i^c and O_i^g, we define

$$\overline{O_i^c} = c_i + O_i^c \quad \text{and} \quad \overline{O_i^g} = g_i + O_i^g,$$

and average the sets of those values instead. It was remarked in the original PEFT paper [12] that including the task cost itself appeared to make little difference, but given the often large disparities between CPU and GPU processing times it seems sensible to include them here.

2.4.2 Processor selection

As discussed in Section 2.3.1, there is arguably much less scope for alternative processor selection rules within the rubric of a priority-based heuristic: ultimately, the only reason not to make the greedy choice is if we expect that doing so will lead to a worse makespan in the end. In this section, we consider how the binary heterogeneity of accelerated platforms can be exploited for making this determination.

Binary lookahead. The HEFT with Lookahead heuristic as described in Section 2.3.1 applies the lookahead step for all processing resources when scheduling a task. For an accelerated platform in which each core of the CPUs is considered to be a distinct resource, this is potentially very expensive, even leaving aside the additional complexity in n induced by the lookahead itself. A straightforward way to reduce this cost would be to only consider some subset of the processors; this will presumably be less effective than the full lookahead but may still be useful compared to the standard earliest finish time rule. An intuitive choice that minimizes the cost and seems sensible for accelerated platforms is to do the lookahead step only for the CPU expected to complete it at the earliest time and the corresponding GPU. We refer to the processor selection phase defined by this rule as *binary lookahead* (BL). Recall that there were two variants proposed in [23] which differ in the kind of average used to compare the different sets of finish times for the child tasks—i.e., the maximum or a weighted mean. Since our aim is to gauge the utility of the binary restriction itself, rather than which of the two performs best, we consider only the former here.

Optimistic lookahead. This idea of comparing only the best CPU and GPU resources also occurs in the PEFT [12] heuristic. Rather than looking ahead through simulation, the idea is to weigh best-case estimates of the future savings we can achieve against the immediate time saved by making the greedy choice. In particular, suppose F_i^c is the expected finish time of task t_i on the fastest CPU and F_i^g the corresponding time on the fastest GPU. Then if

$$F_i^c + O_i^c < F_i^g + O_i^g,$$

we select the fastest CPU, and otherwise the GPU. We call the processor selection rule defined in such a way *optimistic lookahead* (OL). In Section 2.3.2, we described two slightly different ways the optimistic costs could be calculated: either using Eq. (2.10) as in the original heuristic, or using Eq. (2.11) instead. We refer to these variants as OL-I and OL-II, respectively.

GCP and HAL. The optimistic costs used in PEFT are computed recursively so that each value represents a lower bound on the total future costs given that a task is assigned to a processor type. Intuitively, we can see how this is useful as a correction that prevents greedy processor selections obstructing globally optimal performance. However, it is possible that the estimates are too optimistic to actually be useful in making this determination; for example, it was noted in [23] that increasing the horizon for the (simulated) lookahead could be detrimental because of the cumulative distortion from building on optimistic finish time estimates. Therefore we suggest that it may be beneficial to do an optimistic lookahead but keep the horizon local.

In particular, we propose two simple new processor selection rules based on consideration of how each processor type selection affects the best possible finish times of a task's children. For each child task t_k of a generic task t_i, define ϵ_{ik} to be the shortest possible time between when t_i finishes and when t_k does. Suppose t_i is scheduled on a CPU. Then there are two possible values ϵ_{ik} may take: c_k, if t_k is also scheduled on a CPU, and $d_{ik} + g_k$, if it is scheduled on a GPU—i.e.,

$$\epsilon_{ik} \in \{\underbrace{c_k, \ldots, c_k}_{r}, \underbrace{d_{ik} + g_k, \ldots, d_{ik} + g_k}_{s}\},$$

where the multiplicities reflect the number of processor selections for t_k corresponding to each value. However, if t_i is scheduled on a GPU, then we have

$$\epsilon_{ik} \in \{\underbrace{d_{ik} + c_k, \ldots, d_{ik} + c_k}_{r}, \underbrace{d_{ik} + g_k, \ldots, d_{ik} + g_k}_{s-1}, \underbrace{g_k}_{1}\}.$$

Of course, we do not know where the child tasks will actually be scheduled no matter which processor type t_i is scheduled on. However, we can compare the values that we expect ϵ_{ik} to take in the two different cases—i.e., model ϵ_{ik} as a random variable and compute $\mathbb{E}[\epsilon_{ik} \mid t_i \text{ on CPU}]$ and $\mathbb{E}[\epsilon_{ik} \mid t_i \text{ on GPU}]$. Then we can define a processor selection rule by scheduling t_i on the fastest CPU if

$$F_i^c + \max_k \mathbb{E}[\epsilon_{ik} \mid t_i \text{ on CPU}] < F_i^g + \max_k \mathbb{E}[\epsilon_{ik} \mid t_i \text{ on GPU}], \tag{2.21}$$

and the fastest GPU otherwise.

We considered two different ways to compute the expectations. First, we can assume, as when computing upward ranks in HEFT, that all processors are equally likely for each child task—i.e., define the expectations as the arithmetic means of the sets of possible values corresponding to each processor selection for the children. In fact, in this case the selection rule greatly simplifies since it reduces to simply comparing the mean communication cost of each edge assuming that t_i is scheduled on either a CPU or GPU. In particular, if t_i is scheduled on a CPU then the mean cost of the edge (t_i, t_k) is $\frac{s}{q}d_{ik}$ and if it scheduled on a GPU then the mean cost is $\frac{r+s-1}{q}d_{ik}$, so that there is effectively an expected penalty of $\frac{r-1}{q}d_{ik}$ associated with choosing a GPU over a CPU (for each child). This in turn suggests that we should always choose the fastest CPU for each task unless

$$F_i^g + \frac{r-1}{q}\max_{k\in\Gamma^+} d_{ik} < F_i^c,$$

in which case we would choose the fastest GPU. We refer to this processor selection rule as *GPU communication penalty* (GCP).

The obvious issue with the GCP rule is that assuming all processor selections are equally likely is unrealistic. Alternatively, we could compute the expectation by taking the harmonic mean of the possible values that ϵ_{ik} may take, so that smaller values are accorded more likelihood. In this case, we have

$$\mathbb{E}[\epsilon_{ik} \mid t_i \text{ on CPU}] = \frac{qc_k(d_{ik}+g_k)}{r(d_{ik}+g_k)+sc_k}$$

and

$$\mathbb{E}[\epsilon_{ik} \mid t_i \text{ on GPU}] = \frac{qg_k(d_{ik}+c_k)(d_{ik}+g_k)}{rg_k^2+d_{ik}^2+qd_{ik}g_k+sg_kc_k+c_kd_{ik}}.$$

We call the processor selection rule defined by using Eq. (2.21) with these values *harmonic average lookahead* (HAL).

2.5 SIMULATION RESULTS

To evaluate the methods proposed in the previous section, we used a bespoke software simulator which implements the mathematical model described in Section 2.2.1 and therefore facilitates the evaluation of scheduling algorithms for idealized accelerated platforms. Although this model may not capture the full range of real-world behavior—such as processor failure, cache misses, etc—this approach allows us to compare multiple scheduling algorithms and determine how intrinsically well-suited they are for accelerated architectures. Recall from Section 1.4 that the complete source code for the simulation software can be found at the Github repository associated with this thesis[2].

[2] https://github.com/mcsweeney90/thesis-code

2.5.1 Testing environment

We used two different sets of task graphs in our simulations. The first comprised graphs based on a single real application, namely Cholesky factorization, with computation and communication costs generated through timing experiments on a real machine. The second set contained graphs with randomly generated topologies from an existing benchmark [128] and costs likewise generated randomly according to several parameters, in an attempt to cover a wide range of possible applications.

Cholesky DAGs. For $N = 5, 10, 15, \ldots, 50$, we constructed the topology of a task DAG for the Cholesky factorization of an $N \times N$ matrix according to Algorithm 1.1. This gave us 10 topologies with between 35 and 22100 tasks. We assume that all of the matrix tiles are uniform and square, so that the corresponding graph weights are determined by a single *tile size* parameter nb, the number of elements along the vertical (and horizontal) axes of the tiles.

To generate realistic task computation costs on CPU and GPU, we timed the relevant BLAS or LAPACK (CPU) and cuBLAS or cuSOLVER (GPU) routines for 1000 randomly generated matrices of various sizes on a heterogeneous node of a University of Manchester computer cluster [81]. This node comprises four octacore Intel (Skylake) Xeon Gold 6130 CPUs running at 2.10GHz with 192GB RAM and four Nvidia V100-SXM2-16GB (Volta) GPUs, each with 16GB GPU global memory, 5120 CUDA Cores and NVLink interconnect. As stated previously, we regard CPU cores as individual processing resources but GPUs as discrete, so that in particular we timed each routine 1000 times on both a single CPU core and an individual GPU. Table 2.3 summarizes the mean μ and standard deviation σ of the kernel runtimes for tile sizes $nb = 128$ and $nb = 1024$. These sizes were chosen because they roughly represent the lower and upper limits of what is practical for CPU-GPU platforms: the GPU is wasted on smaller sizes and CPUs struggle for larger ones.

A common measure of the spread of a distribution is the *coefficient of variation*, calculated as σ/μ. For the data in the table, the coefficient of variation ranges from less than 0.01 to around 0.1, reflecting the fact that the timing distributions were reasonably tight around the mean. Therefore we follow the usual convention of taking the mean values as the scalar computation cost estimates used for scheduling. Assuming this to be the case, Table 2.4 presents the acceleration ratios of the kernels. As we might expect, the ratios are

Table 2.3: Summary of timing data (in microseconds) for Cholesky factorization BLAS/LAPACK kernels. 1000 timings were observed for each kernel and tile size nb.

	CPU				GPU			
	$nb = 128$		$nb = 1024$		$nb = 128$		$nb = 1024$	
Kernel	μ	σ	μ	σ	μ	σ	μ	σ
`GEMM`	89.9	9.1	41369.0	1063.0	11.3	1.5	446.4	1.9
`POTRF`	141.1	3.1	16219.5	434.3	84.8	0.7	1184.6	1.9
`SYRK`	72.2	2.3	23363.2	603.9	31.9	0.5	419.0	1.5
`TRSM`	75.2	7.9	22206.1	1167.0	44.1	0.5	916.6	3.4

Table 2.4: Acceleration ratios for Cholesky factorization kernels.

Tile size	`GEMM`	`POTRF`	`SYRK`	`TRSM`
128	8.0	1.7	2.3	1.7
1024	92.7	13.7	55.8	24.2

considerably greater for the larger tile size. Moreover, the preference strengths of the different kernel types are broadly what we might expect too, with the more straightforwardly parallel `GEMM` (matrix multiplication) having the strongest preference for the GPU.

The tile size nb also dictates how much data needs to be moved between any two tasks—and therefore the communication costs. Assuming that the entire tile needs to be transmitted in all cases, we calculated rough estimates of the communication delays through the following procedure.

1. Create a matrix of the given tile size with randomly generated `double` elements.
2. For each kernel, randomly generate any other necessary matrices/vectors and copy them to GPU memory.
3. Start the CPU timer.
4. Apply GPU kernel to the matrix and time its execution separately on the GPU.
5. Stop the CPU timer.

The CPU-GPU communication delay is assumed to be the difference between the GPU kernel execution time and the time taken for the entire operation. As with the computation times, we repeated this procedure 1000 times and took the mean in order to get scalar cost estimates. Based on the assumptions of our model, the CPU-GPU communication delay was also taken to be the GPU-CPU and GPU-GPU communication delay as well. It should be emphasized that modern runtimes systems will use much more sophisticated performance models to estimate communication delays than the crude method described here.

The relative size of the computation times and communication delays varies depending on the tile size and kernel type. Typically, we found that the communication delay associated with a task—i.e., the time to move the associated data—fell somewhere between the CPU and GPU execution times, tilting more heavily to the former. But it isn't immediately clear how to quantify the relative amount of computation and communication in heterogeneous scheduling, since there are different possible costs depending on where the tasks are scheduled. Therefore (arithmetic) mean values are typically used instead [131]. In particular, a quantity β called the *communication-to-computation ratio* (CCR) is often defined, computed through $\beta = \overline{w_{ik}}/\overline{w_i}$, for any parent and child pair t_i and t_k, where $\overline{w_i}$ and $\overline{w_{ik}}$ are as defined by Eqns. (2.2) and (2.3) respectively. However, since the quantity $\overline{w_{ik}}/\overline{w_i}$ varies for different task types, even though the amount of data moved is the same for all tasks, in this case it seems more sensible to consider the graph as a whole and define its CCR through

$$\beta = \frac{\sum_i \sum_k \overline{w_{ik}}}{\sum_i \overline{w_i}}, \tag{2.22}$$

i.e., the ratio of the average total communication and the average total compute. Note that since it uses mean values, the CCR depends on the composition of the target platform. In our investigation, we assumed that the number of GPUs was fixed at $r = 32$ and considered two different choices for the number of GPUs, $s = 1$ and $s = 4$ (see below). For tile size 128, CCR values for the Cholesky DAGs were around 0.25 with $s = 1$ and 1.0 for $s = 4$, whereas for tile size 1024 the corresponding values were about 0.01 and 0.05, suggesting that computation is more predominant in that case.

Randomly generated DAGs. As a testbed for benchmarking scheduling algorithms, Tobita and Kasahara [128] proposed the *Standard Task Graph* (STG) set, a large collection of artificial task graphs, which is freely available online at:

`http://www.kasahara.cs.waseda.ac.jp/schedule/index.html`.

The graphs in the STG set were created using several standard methods for randomly generating DAGs, with the intention of capturing as wide a variety of topological features as possible. Therefore, we decided to use their topologies as the basis for a set of randomized task graphs in our own simulation environment.

We followed a similar approach to [19] and [36] to generate computation costs for these DAGs. In particular, for each task we sampled its CPU and GPU execution times from gamma distributions with means 15 and 1, respectively. This corresponds to an expected acceleration ratio of 15, which is broadly in line with what we observed when benchmarking BLAS kernels. For both distribution choices, we assumed that the coefficient of variation is equal to 1; this is somewhat larger than we observed for the BLAS kernels, but we wished to consider a wider range of task acceleration ratios for these graphs. Unlike in [19] and [36], we also needed to generate communication delays. To do this, we used the following procedure, which takes a parameter β that represents a target CCR, as defined in the previous section, for the task graph as input.

1. After setting computation costs for task t_i, compute average $\overline{w_i}$ using Eq. (2.2).

2. Sample ℓ_i uniformly at random from the interval $(0, \frac{2n\beta}{v})$, where v is the total number of edges in the DAG.

3. For all $k \in \Gamma_i^+$ set the nonzero communication cost d_{ik} as

$$d_{ik} = \frac{\ell_i q^2 \overline{w_i}}{s(2r+s-1)}.$$

By choosing d_{ik} in this way we ensure that the target CCR for the graph is approximately met, assuming that $\overline{w_{ik}}$ is computed in the manner defined by Eq. (2.3) (i.e., arithmetic mean). If we set $\ell = \frac{n\beta}{v}$ for all tasks, then the target CCR would be achieved exactly, however we introduced some randomness so that communication delays were not the same for all edges. Although this meant that the CCR is only approximate, it was usually very close to the target, especially for larger graphs. Moreover, the order of magnitude differences between the values of β considered (see below) dwarf the uncertainty from the cost-setting procedure.

The STG comprises several subsets, each corresponding to a different number of tasks and containing 180 graphs; since the Cholesky factorization graphs vary in size, we used only the subset with $n = 1000$ tasks. For each topology, we chose $\beta \in \{0.01, 0.1, 1.0, 10.0\}$ and generated 10 different sets of costs for each. Altogether, this means that our randomized graph set effectively comprised $180 \times 4 \times 10 = 7200$ DAGs. (In fact, the size of the graphs in the STG does not include the entry and exit tasks, so that the graphs in the set actually had 1002 tasks. However, we use the rounder figure for clarity.)

Target platforms. We considered only two different target platforms: one comprising $r = 32$ CPU resources and $s = 1$ GPU, and the other $r = 32$ CPU resources and $s = 4$ GPUs. These were intended to reflect the platform that we used to benchmark the Cholesky BLAS kernels—i.e., four octacore CPUs and up to four GPUs—which is fairly typical for an accelerated node in HPC today. The two different values $s = 1$ and $s = 4$ were used in order to evaluate how the number of GPUs impacts the scheduling problem. Since we kept the number of CPUs fixed at $r = 32$, we can therefore regard $s \in \{1, 4\}$ as a variable which defines the target platform.

2.5.2 Performance metrics

When evaluating how good a schedule π with makespan $|\pi|$ is for a given task graph, we want to know how close it is optimal. Therefore the metric that we would ideally like to use is the ratio $|\pi|/|\pi^*|$, where π^* is the optimal schedule. Of course, actually computing π^* tends to be impossible for all but the smallest graphs, so some alternative must be used instead. The natural choice is the tightest possible lower bound on the makespan. In much of the recent literature [8], [19], the relaxed solution to the assignment LP (see Section 2.3.3) is used, but, as noted earlier, this is difficult to even formulate for the communication model that we follow here and would be expensive to solve at any rate. Therefore we used a cheaper bound Ω computed by combining two classic bounds in the following manner.

First, we have the work bound, defined as the total amount of work that must be done divided by the number of processors. Of course, the total amount of work done depends on the schedule we follow, but we know that the work done for each task is bounded below by the smallest of the two possible computation costs, so that

$$B_W = \frac{1}{q} \sum_{i=1}^{n} \min(c_i, g_i)$$

gives a lower bound on the makespan. The other classic bound on the makespan is the critical path bound, the length of the longest path through the task graph. Again, this is unknown until the graph is actually scheduled, but we have already seen a lower bound for this quantity: the smallest of the two optimistic costs, as defined in the PEFT heuristic [12], for the entry task t_1. (As discussed in Section 2.3.2, the version from the original paper is not a true lower bound since it uses average values but this can be fixed by using the known values instead.) In other words,

$$B_P = \min\left\{c_1 + O_1^c, g_1 + O_1^g\right\}$$

also gives a lower bound on the makespan, where O_1^c and O_1^g are as defined by Eqns. (2.12) and (2.13), respectively. Bringing these together, we define

$$\Omega = \max\{B_W, B_P\},$$

so that the ratio $|\pi|/\Omega$ is a measure of how close the schedule π is to the lower bound. We refer to this as the *schedule length ratio* (SLR), following a similar quantity used in [131], although how we calculate the makespan lower bound differs. Note that the SLR is bounded below by one.

An alternative performance metric often used in the literature is the *speedup*. It is in some sense contrary to the SLR, in that while the SLR indicates how much worse the schedule makespan is than the optimal, the speedup suggests how well the schedule does to one that is expected to be much worse. Define the *minimal serial time* (MST) to be the shortest time in which the entire task graph can be executed on a single processor; in our case, this is given by $\min(\sum_i c_i, \sum_i g_i)$. Then the speedup is defined as the ratio of the MST and the schedule makespan, so that a large speedup suggests a better schedule than a smaller one. We prefer to use the SLR rather than the speedup, with the exception of when the speedup is useful for identifying when a schedule is so poor that it can be regarded as having *failed* completely: namely, when the speedup is less than one.

Whereas the SLR and speedup both quantify how good a schedule is compared to some reference, in our investigation we often computed several different schedules for the same task graph and wanted to evaluate how good they were relative to one another. A useful metric for this is the *percentage degradation* (PD), defined, for a given schedule, as the percentage increase in makespan relative to the shortest computed schedule for that task graph. Note that the PD is therefore bounded below by zero and a low PD is better than a high one.

2.5.3 Task prioritization

As there were so many different task prioritization schemes described in Section 2.4.1, to compare them fairly we assumed that the standard earliest finish time (EFT) rule was used for processor selection in all cases (with the partial exception of the autopsy method, which only uses EFT to choose between processors of the assigned type). Although there is clearly some interplay between the two phases of the priority-based framework, in general one would expect a good prioritization scheme to perform better than a bad one, no matter which processor selection rule is followed. Note that, since the autopsy method is conceptually different from the other task prioritization phases, we considered it separately.

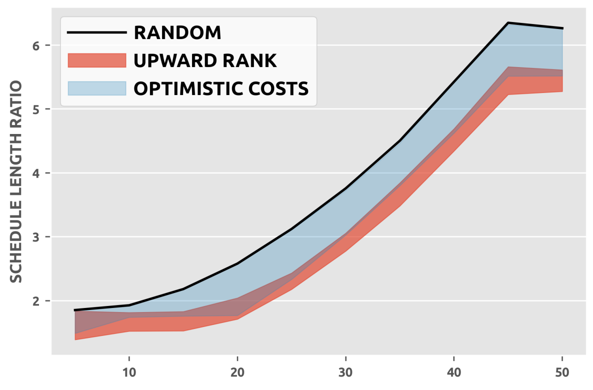
Upward ranking and optimistic costs. Fourteen different averaging schemes within the upward ranking rubric were listed in Table 2.2. Ten of these can also be applied to the optimistic costs since there is no need to distinguish the four "simple" averages in that case. This gives us a total of 24 different task prioritization phases. Where necessary, we will refer to a specific task prioritization scheme as A-U or A-O, where A is the name of the averaging scheme and the letter after the hyphen indicates whether the average is used for computing upward ranks (U) or applied to the set of optimistic costs (O); for example, M-U means the task prioritization defined by using arithmetic mean averages for all graph weights and then computing the upward rank. As a baseline for comparison, we also considered a *random* task prioritization phase, defined by scheduling tasks according to a topologically sorted list generated via a depth-first search algorithm from [80]. We call it random in the sense that it can be viewed as a random sample from the set of all valid topological sorts. Since it is computed without any consideration of the graph weights, clearly we should hope that all of the other prioritizations are superior to this simple alternative. Note that since runtimes were very similar for all of the task prioritization phases—even including the time for computing the optimistic costs if necessary—this comparison is concerned only with schedule quality.

First, we considered the Cholesky graphs. In general, most of the task prioritizations did well, at least in comparison with the random phase. Figure 2.1 shows the range of SLRs achieved by the two different approaches—upward ranking or optimistic costs—where the shaded regions indicate the ranges between the best and worst average types for each graph. We see that the best prioritization phases of both types are always clearly superior to the random phase but the worst, especially for the optimistic costs, are typically no better. In fact, most of the averaging schemes for the optimistic costs usually did well; the problem is that the preference-based R, D, NC and SD schemes were almost always very bad when applied to the optimistic costs, with the odd exception that NC-O was the outright best of all the prioritization phases for the graphs with $nb = 1024$. Moreover, we see from the figure that the worst rankings are often precisely as bad as the random one. This is no coincidence and reveals why they did not do well: our implementation uses the random topological sort as an index, with ties broken according to their position in that list. The worst phases are so similar to the random one because too many tasks are given similar priorities. Aside from the unusually strong performance of NC-O for $nb = 1024$, the upward ranking phases were generally superior to their optimistic cost counterparts, especially for the smaller tile size.

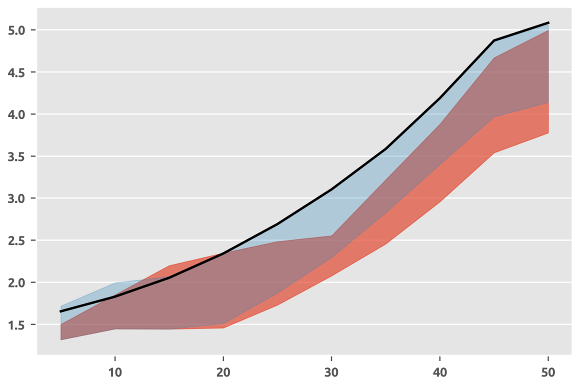
Another interesting observation from Figure 2.1 is that the magnitude of the SLRs differs significantly depending on the tile size and the number of GPUs available. Unfortunately, it isn't clear whether this is due to all of the prioritization phases doing objectively better or worse in the different cases, or whether the problem is that the makespan lower bounds used to compute the SLR are too optimistic for certain parameter choices (i.e., the gap between the lower bounds and the optimal *feasible* makespan varies).

Figure 2.2 shows the mean percentage degradation (MPD) obtained by the 24 prioritization phases for the Cholesky graphs. The most immediate takeaway is that the preference-based averages for the optimistic costs were consistently much worse than the others (again, with the exception of NC-O for $nb = 1024$). Beyond that, it is hard to draw any firm conclusions as to which of the rankings is best. The standard HEFT M-U phase did well overall, as did the MD-U, W/SW-U, HM/SHM-U and SD-U schemes. But, as already noted, all of these were dominated by NC-O for $nb = 1024$. Indeed, considering the entire set of 40 Cholesky factorization graphs, all of the rankings but R-U obtained the best schedule at least once (including ties). This suggests that deciding which task prioritization scheme is likely to give the best schedule is very much a local problem for which the answer depends on a wide variety of factors.

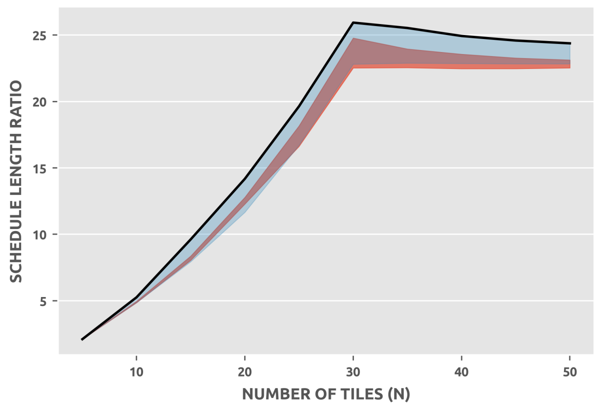
Our conclusions were similar for the STG set. As can be seen from Figure 2.3, the few overall trends that we observed for the Cholesky graphs also largely held across the set



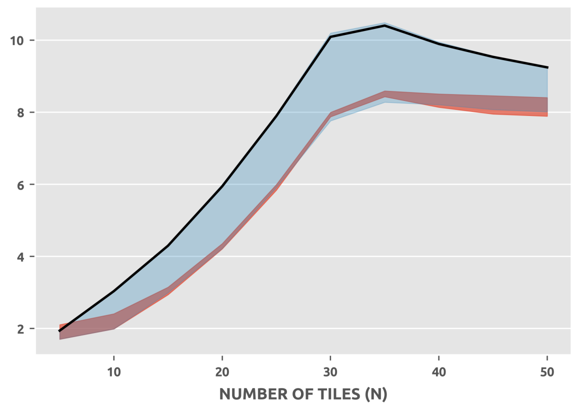
(a) $s = 1$, $nb = 128$.



(b) $s = 4$, $nb = 128$.

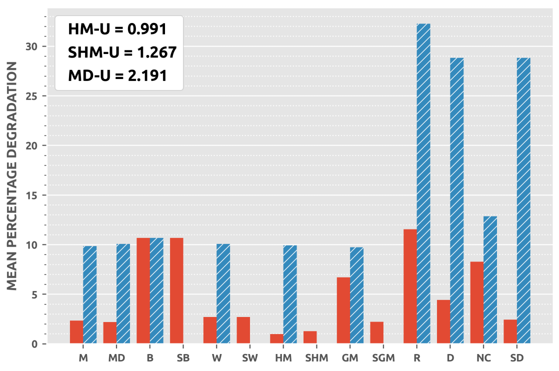


(c) $s = 1$, $nb = 1024$.

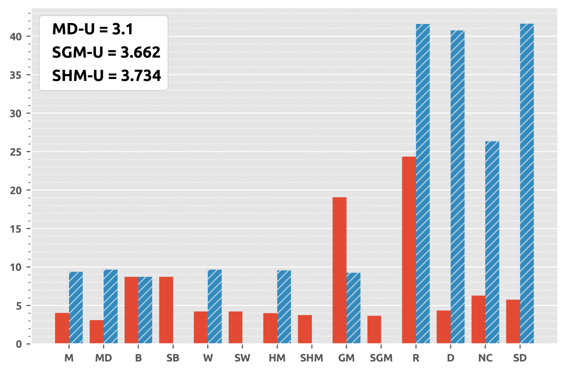


(d) $s = 4$, $nb = 1024$.

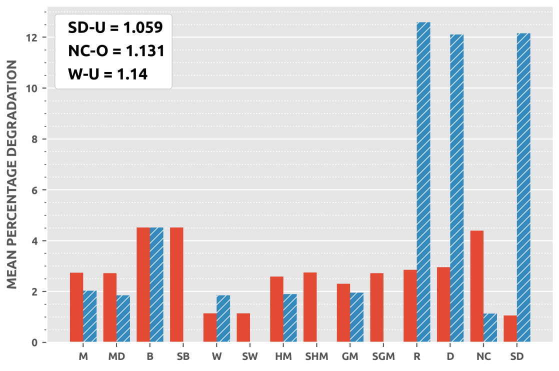
Figure 2.1: Schedule length ratios (SLRs) of task prioritization schemes for Cholesky graphs with different combinations of s (number of GPUs) and nb (the tile size). Black line indicates the random prioritization. Red shaded region represents the difference between the best and worst upward ranking averaging schemes for each graph and the blue region likewise for the optimistic cost averages. Recall that N is the number of tiles along both axes of the matrix.



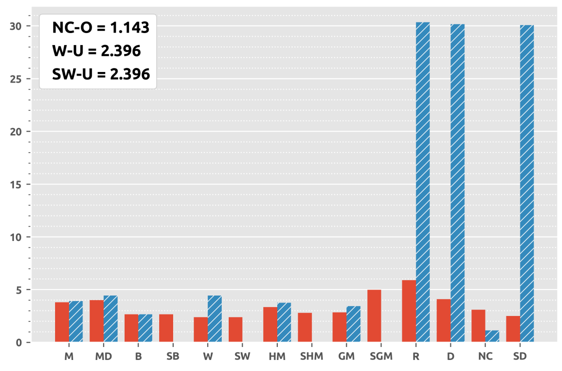
(a) $s = 1$, $nb = 128$.



(b) $s = 4$, $nb = 128$.



(c) $s = 1$, $nb = 1024$.



(d) $s = 4$, $nb = 1024$.

Figure 2.2: Mean percentage degradation (MPD) of task prioritization schemes for Cholesky graphs with different combinations of s (number of GPUs) and nb (the tile size). Solid red bars indicate upward ranking and striped blue bars optimistic cost averages; recall that the SB, SW, SHM and SGM averages are not defined for the optimistic costs. Legends identify the three best schemes.

as a whole. For example, R-O, D-O and SD-O were again the worst, although this time NC-O, B-O and SB-U were equally as bad. There was also a more consistent advantage to upward ranking rather than the optimistic costs; indeed, the former were better than the latter for every single average type. The rankings which were usually among the best for the Cholesky graphs also did well here. W-U achieved the smallest MPD, followed by SW-U, SD-U, M-U and HM-U. The differences between them were relatively small, however, reflecting the fact that again there was a lot of local variation and almost all of the average types did well in some cases and badly in others.

This is highlighted by Figure 2.4, which shows the MPDs of the ranking schemes when the STG set is broken according to different choices for the parameters s and β. Note that each of the bar charts therefore represent subsets of $7200/8 = 900$ graphs. We see from the figure that although many of the same average types recur in the lists of the best three rankings, it would be very difficult to anticipate the best average for a given set of parameters. For example, HM-U is never among the top three except for $s = 4$ and $\beta = 1.0$, when it is comfortably the best. Moreover, although its 1% advantage in MPD compared to the next best may not seem like much, this represented a 75% probability of returning a better schedule than M-U, the standard HEFT task ranking phase, suggesting that it should be the default choice under such parameter regimes. We can certainly make intuitive arguments as to why certain average types may perform well when, for example, our target platform boasts many GPUs—but there are typically other averages for which similar arguments can be made. This means that choosing which average type to use in a priority-based heuristic is a decision that is best made on a case-by-case basis, perhaps informed by benchmarking experiments similar to those conducted here.

Of all the subgraphs, the most divergent behavior is seen in Figure 2.4g: unlike every other subset, the optimistic cost averages are better than their upward ranking counterparts almost across the board. Furthermore, the MPDs of every method are considerably greater than for other parameter choices. In fact, the difference with the other subsets is even more pronounced than the figure indicates. In particular, we found that, for these graphs, all of the task prioritization schemes *failed*—i.e., returned a schedule whose makespan was greater than the minimal serial time—a high percentage of the time. The upward ranking schemes were particularly bad, all failing for at least 80% of the graphs, whereas the optimistic cost schemes recorded failure rates of around 50%, explaining why

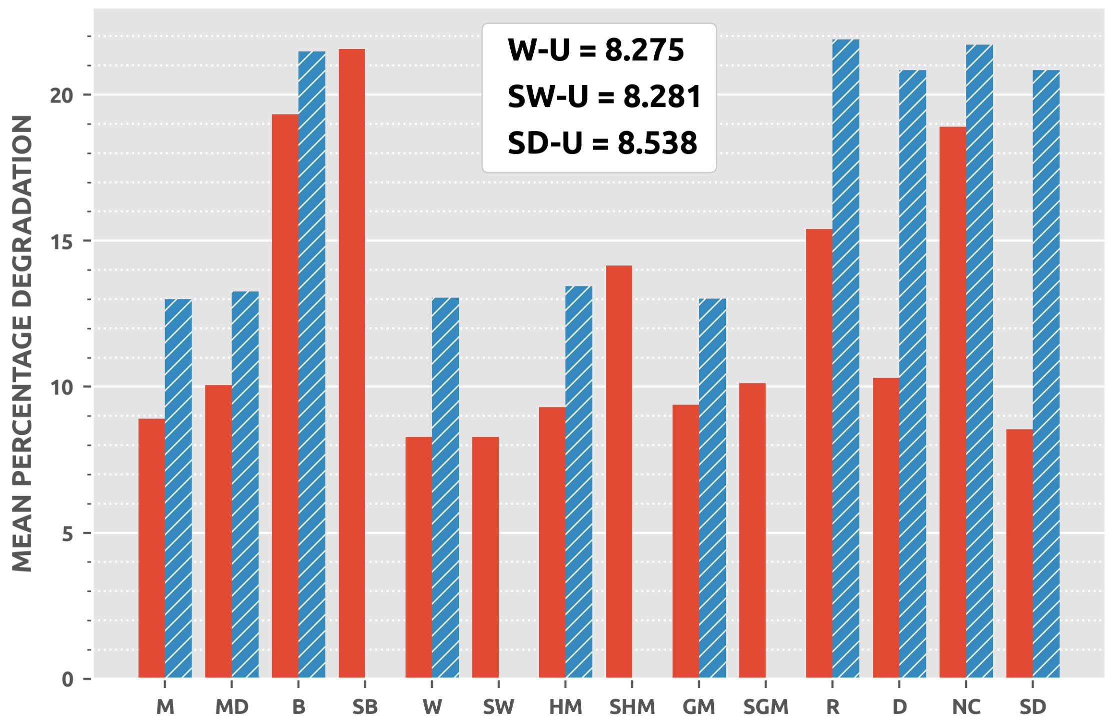
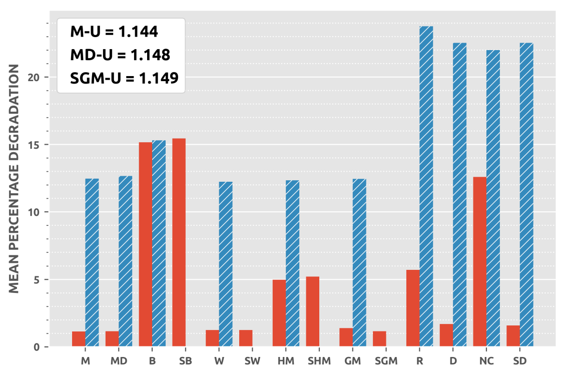


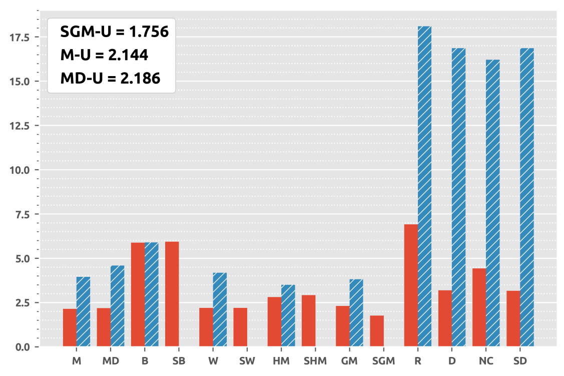
Figure 2.3: Mean percentage degradation (MPD) of task prioritization schemes for entire STG set. Solid red bars indicate upward ranking and striped blue bars optimistic cost averages; recall that the SB, SW, SHM and SGM averages are not defined for the optimistic costs. Legend identifies the three best schemes.

they performed better on average. This might lead one to suspect that optimistic costs are more robust against failures. However, failure rates were also extremely high for the graphs in Figure 2.4h—and in that case the corresponding figures were about 60% for the upward ranks and 70% for the optimistic costs.

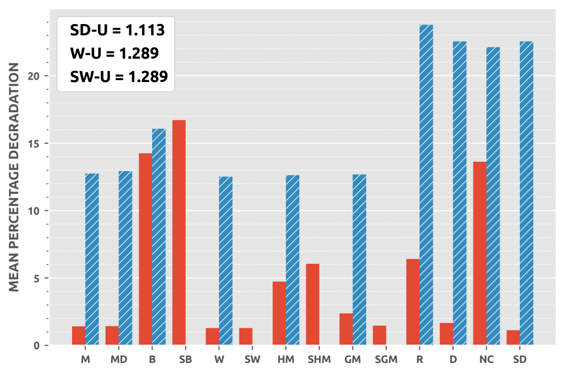
Although there were a small (usually ≈ 1%) percentage of failures for $\beta = 1$, the overwhelming majority occurred for $\beta = 10$, indicating that high communication is the biggest risk factor. In fact, the root cause is the EFT processor selection rule, which does not consider future communication penalties that may be incurred by greedy choices. This is true for all values of β; the problem is simply more pronounced for the largest because the communication penalties are proportionally larger in that case. Given this, and the fact that failure rates were high for all of the many different prioritization schemes that we considered, it seems unlikely that any alternative task prioritization alone can fully remedy this issue, although perhaps this should be investigated in the future.



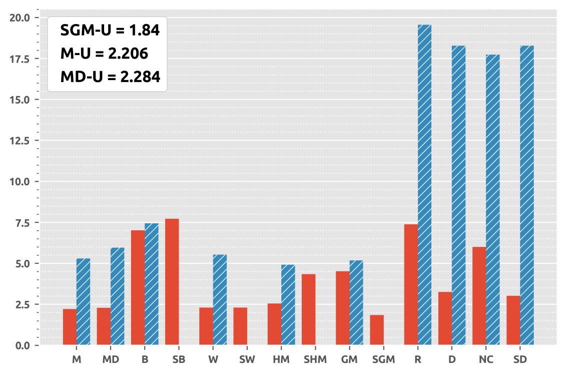
(a) $s = 1, \beta = 0.01$.



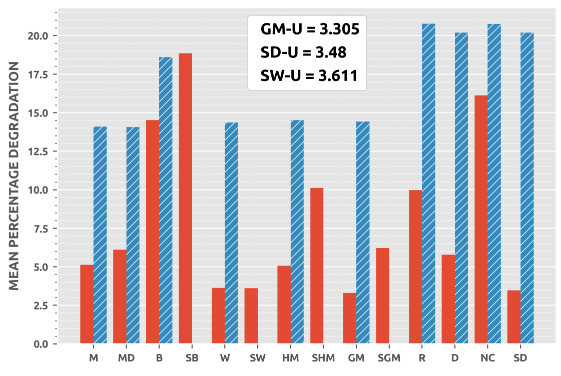
(b) $s = 4, \beta = 0.01$.



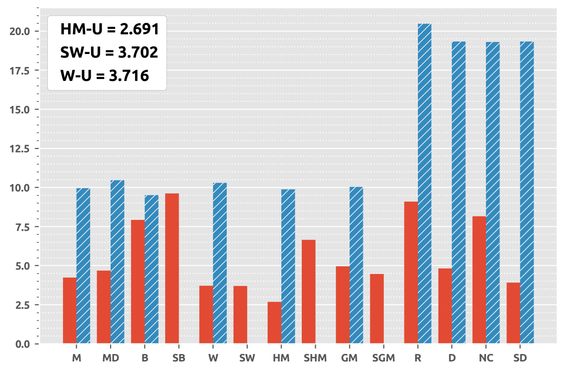
(c) $s = 1, \beta = 0.1$.



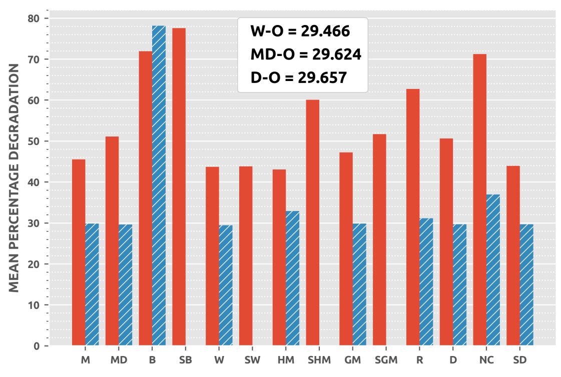
(d) $s = 4, \beta = 0.1$.



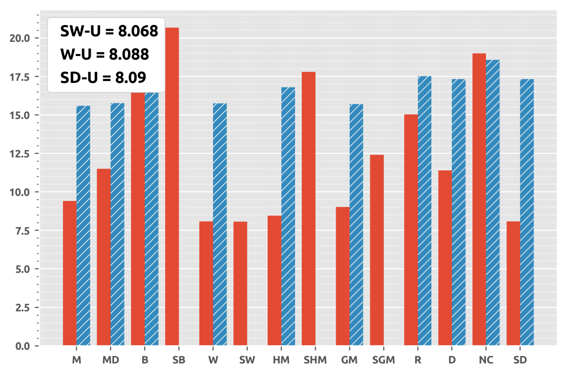
(e) $s = 1, \beta = 1.0$.



(f) $s = 4, \beta = 1.0$.



(g) $s = 1, \beta = 10.0$.



(h) $s = 4, \beta = 10.0$.

Figure 2.4: Mean percentage degradation (MPD) of task prioritization schemes for STG set with different combinations of s (number of GPUs) and β (the CCR). Solid red bars indicate upward ranking and striped blue bars optimistic cost averages; recall that the SB, SW, SHM and SGM averages are not defined for the optimistic costs. Legends identify the three best schemes.

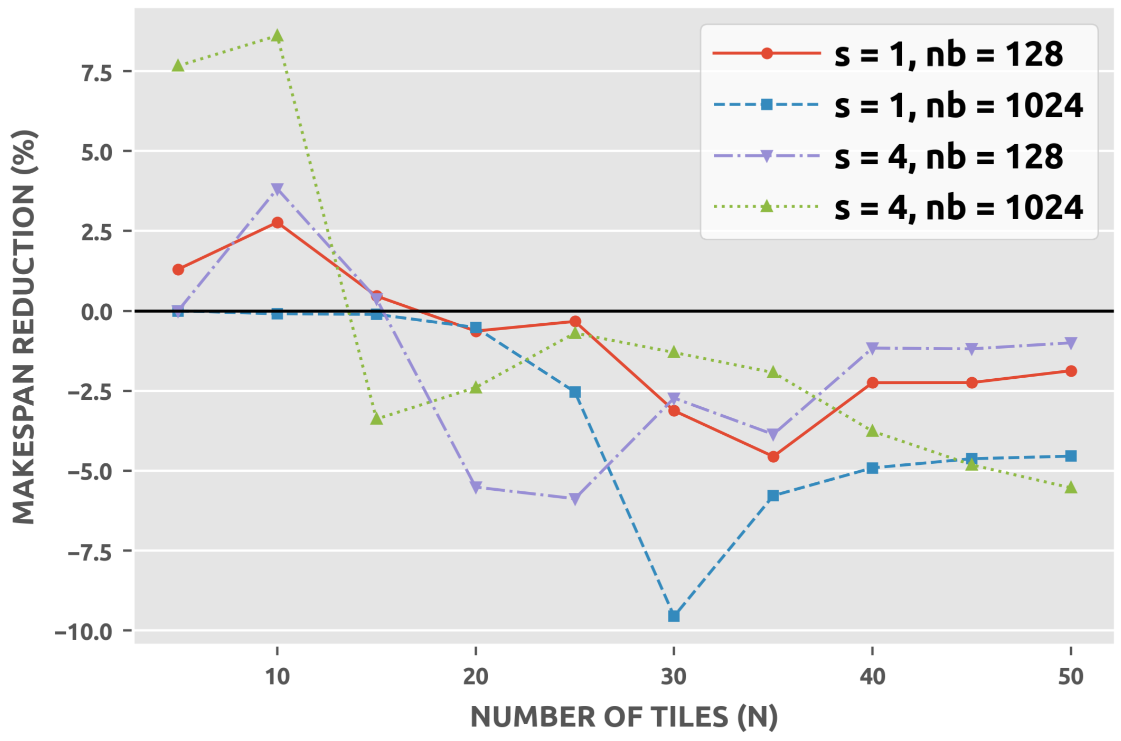


Figure 2.5: Reduction in makespan achieved by applying the autopsy method to HEFT schedules for Cholesky graphs with different combinations of s (number of GPUs) and nb (the tile size). Black horizontal line indicates equality, so that data points beneath represent a *negative* reduction—i.e., the new schedule is worse than the old one.

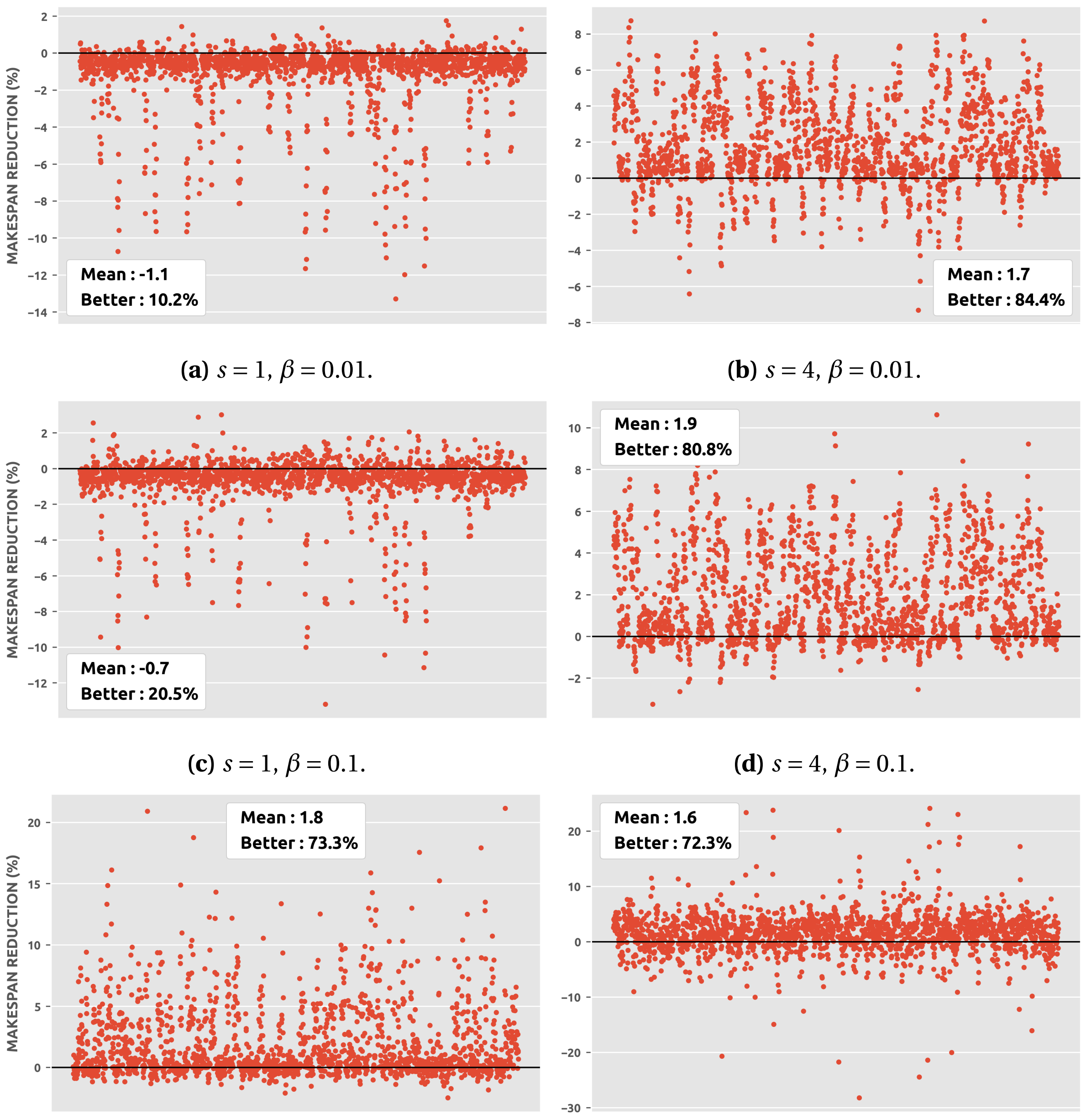
Autopsy. Unlike the averaging-based task prioritization schemes, the autopsy method works by taking an initial schedule as input. To determine if the method is viable, we considered only schedules computed via the standard HEFT algorithm—i.e., M-U task prioritization phase with EFT processor selection rule—for this step. Figure 2.5 shows the makespan reduction achieved by the autopsy method, as a percentage of the original schedule makespan, for the Cholesky graphs. The big takeaway is that the majority of the time there were no reductions and the autopsy method actually led to a *worse* schedule than the original. This is surprising: one would expect that more "accurate" upward ranks would lead to superior performance. Furthermore, increasing the number of GPUs did not seem to improve performance either; again, this may confound expectations, given that the assignment followed in the selection phase is in some sense less restrictive.

The autopsy method actually did well for the smallest Cholesky factorization graphs, so we repeated the experiments for the graphs from the STG set (which have $n = 1000$ tasks) to establish if size is indeed a contributing factor. Figure 2.6 illustrates the makespan reductions for different values of s (the number of GPUs) and β (the CCR). Note that, although it did on average improve the HEFT schedule for $\beta = 10$, the autopsy method still failed for a very high percentage of such graphs, so that data is omitted. We see from the figure that for some parameter choices the autopsy method is considerably more likely to improve on the HEFT schedule—but for others it is again likely to produce a worse schedule. It is not obvious why this is the case. One possible explanation is that strictly following the previous assignment is too inflexible unless processor contention can be safely ignored. This would help to explain why performance deteriorated as the Cholesky graphs grew (more tasks leading to greater contention) and why the probability of producing a better schedule was higher for the randomly generated graphs with $s = 4$. But it is not obvious how this would account for the fact that, when $s = 1$, the autopsy method drastically improved for $\beta = 1$ compared to smaller values. Whatever the cause of its unexpected behavior, it should be noted that, when the autopsy method did do well, it was more likely to improve on HEFT than even the best of the alternative average types. This suggests that, when we suspect it may perform well (e.g., there are lots of GPUs), it is worthwhile, although the effective doubling of the runtime must of course also be considered.

2.5.4 Processor selection

Five processor selection rules were described in Section 2.4.2: binary lookahead (BL), the two variants of optimistic lookahead (OL-I and OL-II), GPU communication penalty (GCP), and harmonic average lookahead (HAL). We compared these with two other selection rules: EFT, the standard greedy processor selection rule, and NC, the rule used in the HEFT-NC heuristic [114]. For the latter, we used the suggested cross threshold $X = 0.3$. As in the previous section, we took a modular approach and used the M-U prioritization phase—upward ranks with (arithmetic) mean averages—in all cases. One hopes that a good processor selection rule will be more effective than a bad one, no matter how the task priorities are computed; exploratory experiments suggested that this was largely true so long as the different task prioritization phases performed similarly well, reinforcing the need to optimize both phases of the priority-based framework.

For the Cholesky factorization DAGs, we found there was little difference between any of the processor selection rules for tile size $nb = 1024$, no matter how many GPUs the target platform hosted; the percentage degradation for each was less than 1% on average



(a) $s = 1$, $\beta = 0.01$. **(b)** $s = 4$, $\beta = 0.01$.

(c) $s = 1$, $\beta = 0.1$. **(d)** $s = 4$, $\beta = 0.1$.

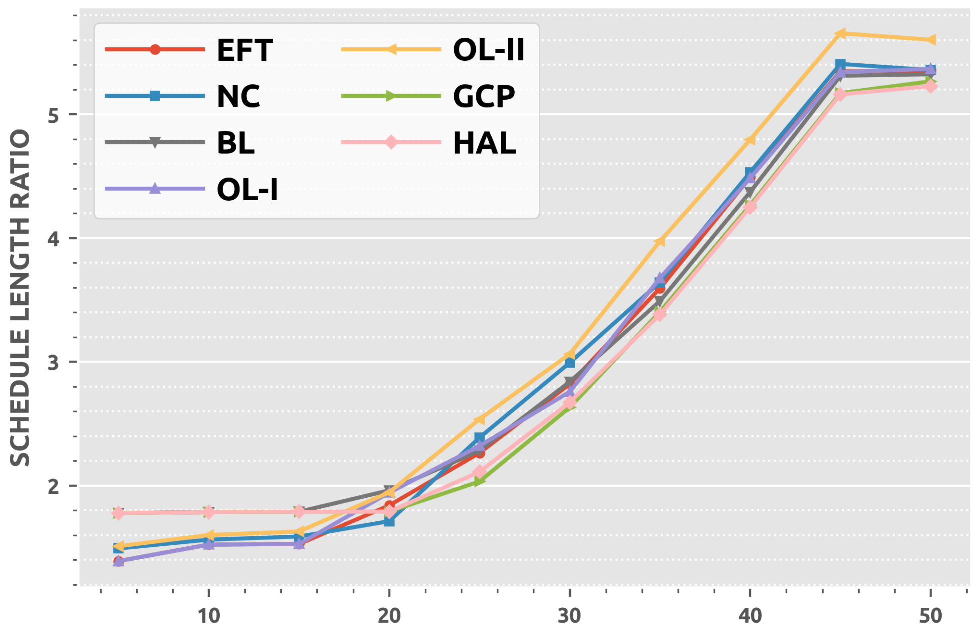
(e) $s = 1$, $\beta = 1.0$. **(f)** $s = 4$, $\beta = 1.0$.

Figure 2.6: Reductions in makespan achieved through the autopsy method for STG set with different combinations of s (number of GPUs) and β (the CCR). Black horizontal lines indicate equality so that values above imply superior performance. Legends indicate the mean percentage reduction and the percentage of graphs for which the autopsy method gave a better schedule. Subsets with $\beta = 10$ are omitted because both the HEFT and autopsy schedules were objectively poor.

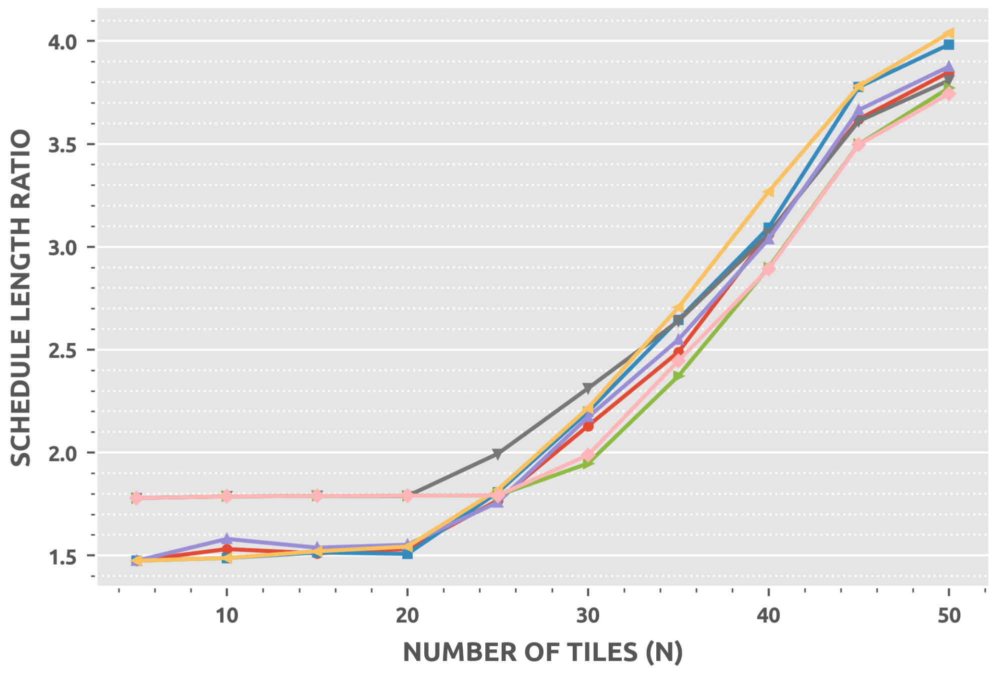
and almost never greater than this. Overall, the GCP rule was most likely to be the best but the advantage over the others was very small. This similar performance is likely because of the greater task acceleration ratios and relatively small communication delays for this tile size: most tasks are assigned to the GPUs in any event and the processor selection rules considered here do not distinguish between processors of the same type.

Results were more interesting for the smaller tile size. Figure 2.7 shows how the schedule length ratio varied with N, the number of tiles along both axes, for each rule. We see some variable behavior; for example, HAL and GCP are among the worst for smaller graphs but always the best for larger ones. In fact, the standard EFT rule was actually the best on average across the entire set since it was the most consistent, always falling in the middle of the pack. With regard to the two optimistic lookahead variants, there is more difference between them than we may expect, given how similar they are, but the larger issue is that both are inferior to EFT overall. Likewise, the NC rule does not appear to actually improve on EFT either, despite being specifically designed for that purpose. This could well be because the cross threshold we use is not suitable for either platform, but without conducting an exhaustive tuning step beforehand it isn't clear what value should be used, a clear downside of this approach. The BL rule is at least competitive with EFT but given the greater complexity one would hope for more clear-cut gains; for our own implementation, we found that runtimes could be more than 100 times longer than the other selection rules (which were very similar to one another).

We repeated the comparison for the STG set, although unfortunately runtime constraints prevented our including the BL rule since it was so much more expensive than the others. Figure 2.8 presents the mean percentage degradation (MPD) achieved by each selection rule. Note that unlike in Figure 2.6 we have included the data for $\beta = 10$ here. However, it was still the case that all of the selection rules did badly, as can be seen by the fact that the MPD of all the rules was greater than 100% for $s = 1$. Considering all the graphs with $\beta = 10$, the failure rate for most of the rules was still around 65–70%; OL-II was the best at about 45% but that is still unacceptably high. Altogether it seems safe to conclude that none of the selection rules considered here really perform well when communication is high. Since the minimal serial time (MST) is usually close to the optimal schedule makespan in such cases, a simple alternative rule would be to follow OL-II until the point at which the current makespan exceeds the MST and then reschedule all tasks



(a) $s = 1$.



(b) $s = 4$.

Figure 2.7: Schedule length ratios (SLRs) of processor selection rules for Cholesky factorization graphs with tile size $nb = 128$ and different values of s (the number of GPUs).

on the fastest processor instead. This would at least avoid failures, although it would only actually improve on the best serial schedule just over half the time. It would be useful in the future to investigate whether more sophisticated techniques such as task duplication are capable of more effectively preventing failures (see next section).

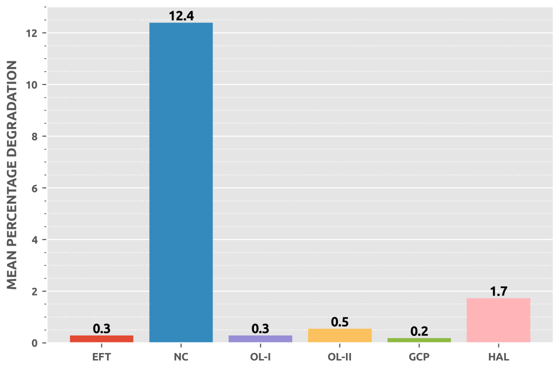
We see the same kind of granularity in Figure 2.8 that we have seen in previous sections. For example, GCP was the best rule for the two smallest β values but was the worst for $\beta = 1$. The default EFT rule performed well and (ignoring $\beta = 10$) was always either the best or second best. NC was much worse than the others for the two smallest β values and mediocre otherwise. HAL was also typically among the worst. Apart from $\beta = 10$, OL-II was consistently worse than OL-I, perhaps because, although the bound doesn't technically hold, the use of average values in OL-I more accurately reflects future schedule costs.

2.6 CONCLUSIONS AND FUTURE WORK

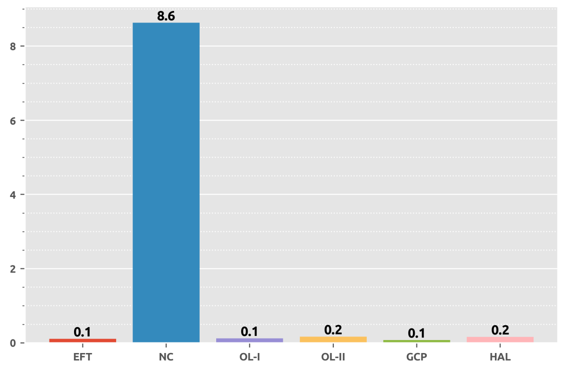
At a high level, our most abiding conclusion from the investigation described above is that there is no one-size-fits-all task prioritization phase or processor selection rule that should always be used for priority-based scheduling on accelerated platforms. Rather, which is most suitable depends on many different factors, such as the number of GPUs and the relative amounts of computation and communication in the task graph. Aside from this overarching impression, we also drew the following more specific conclusions.

1. With regard to which average types should be used for computing task priorities, our experience largely echoes that of Zhao and Sakellariou [147]: significant gains can be made by choosing the best average type but actually identifying which will actually be the best for a given DAG and target platform is difficult. A few overall trends were apparent. For example, upward ranking was generally superior to averaging optimistic costs. Considering the experiments as a whole, the best task prioritization phases on average were W/SW-U, M-U and SD-U. However, alternatives were clearly superior in specific situations, such as HM-U for the randomly generated DAGs with $s = 4$ and $\beta = 1$.

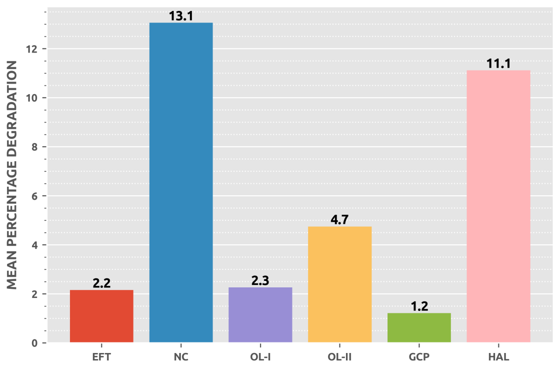
2. The autopsy method appears to have a high probability of improving an existing schedule when there are an adequate number of GPUs and the task graph is not too



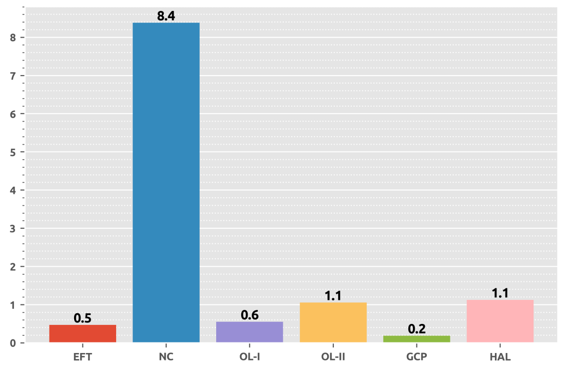
(a) $s = 1$, $\beta = 0.01$.



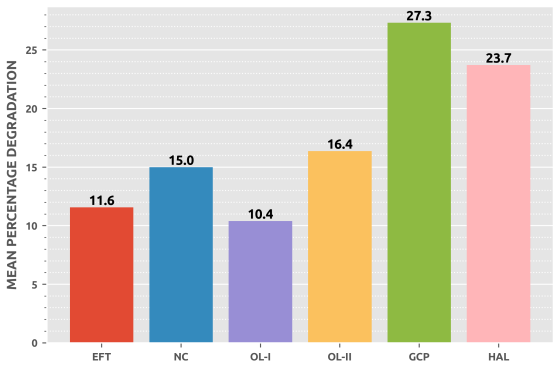
(b) $s = 4$, $\beta = 0.01$.



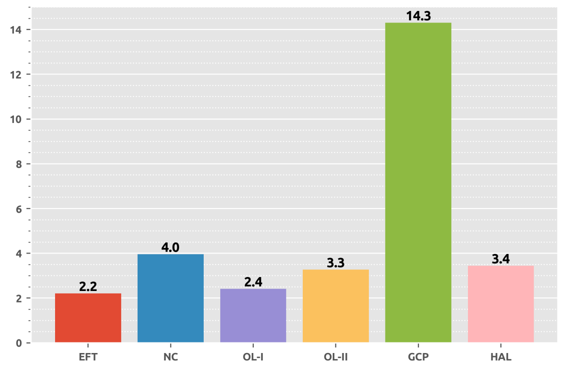
(c) $s = 1$, $\beta = 0.1$.



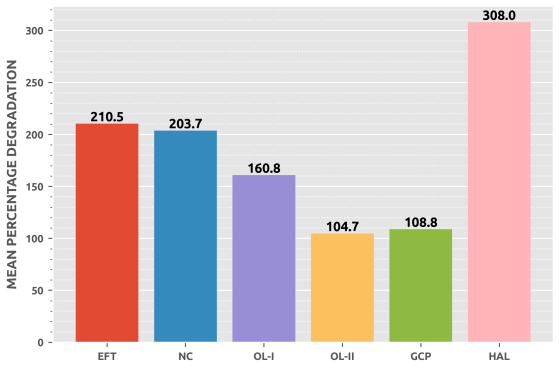
(d) $s = 4$, $\beta = 0.1$.



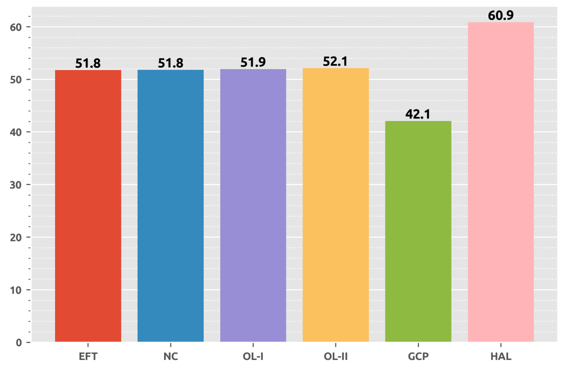
(e) $s = 1$, $\beta = 1.0$.



(f) $s = 4$, $\beta = 1.0$.



(g) $s = 1$, $\beta = 10.0$.



(h) $s = 4$, $\beta = 10.0$.

Figure 2.8: Mean percentage degradation (MPD) of processor selection rules for STG set with different combinations of s (number of GPUs) and β (the CCR).

large. For example, it successfully returned a better schedule than HEFT for about 75% of the randomly generated graphs when the target platform comprised 4 GPUs. However, it appears to actually be counterproductive for large graphs and when there are few GPUs. We suspect that this is due to rigidly following the previous task assignments when processor contention becomes relatively more important; further investigation with large task graphs for applications other than Cholesky factorization may be useful in the future.

3. The default EFT processor selection rule does well when communication costs are relatively small but can struggle as they increase. When communication is high, this can even lead to entirely useless schedules. Unfortunately, with the partial exception of OL-II, none of the alternative selection rules considered here were capable of compensating in such cases either.

4. The simple new selection rule GCP outperformed EFT for the largest Cholesky graphs and the randomly generated graphs when communication costs were relatively low. HAL also did well for the largest Cholesky graphs but was less impressive in all other cases.

5. The NC rule did not appear to be superior to, or even competitive with, EFT or any of the other rules considered, contradicting results in [114] to an extent, although it is noted there that performance may vary depending on the cross threshold parameter; results may well be more positive if guidelines for selecting a good cross threshold can be established.

6. OL-I, the processor selection rule used in PEFT [12], was generally superior to the modified version OL-II proposed here when communication costs were low, although still worse than EFT. On the other hand, OL-II was more effective when communication was high, presumably because it uses true lower bounds on the remaining schedule costs and is therefore a more accurate "correction" in such cases.

In addition to those topics identified above, we suggest that the following may be interesting directions for future research.

1. Alternative communication models should be considered. An obvious choice would be to investigate the effect of varying the size of the GPU-GPU communication delays relative to the delays between CPU and GPU, since they were assumed to be identical here but GPU-GPU delays can in theory be much smaller. Furthermore, contention for communication resources should also be taken into account.

2. Including energy costs when scheduling is increasingly vital, so that is a natural next step. One straightforward approach suggested earlier would be to combine the autopsy method with existing rescheduling heuristics such as LOSS [108], but ideally the latter step should be optimized specifically for CPU/GPU as well.

3. Task duplication can often be useful in priority-based heuristics, both for reducing the makespan [76] and improving reliability [125]. Moreover, it may be useful in reducing the failure probability when communication is high. As noted in Section 2.3, the practical issue tends to be controlling the amount of duplication so that the application runtime (and energy expenditure) remains low. It should therefore be investigated whether the unique properties of accelerated platforms can be exploited to reduce the additional costs.

Chapter 3

The critical path in heterogeneous scheduling

We saw in the previous chapter that there are many different ways that the *critical path* can be estimated in heterogeneous scheduling. Furthermore, different scheduling heuristics use these critical path estimates in different ways. For example, in HEFT, critical paths from all tasks to the sink are calculated in order to prioritize the tasks, whereas in CPOP a single path which is expected to be critical is identified and the tasks which lie on it are all scheduled on the same processor. In this chapter, we extend ideas that were introduced previously for accelerated platforms and consider many different ways that the critical path can be approximated for generic—i.e., not just CPU and GPU—heterogeneous scheduling problems. We cannot expect that any of the alternatives will always be superior to others, so we attempt to identify through simulation which choices are most useful in a wide variety of different situations.

3.1 Generic scheduling model

As in the previous chapter, here we consider *offline* scheduling only and focus on minimizing the schedule makespan. Most of the assumptions from Section 2.2.1 therefore still hold. However, we now assume that there are multiple different processor types, rather than just CPUs and GPUs. Therefore we will use the more general notation that was introduced in Section 2.2.1, rather than that specialized for accelerated platforms. For example, we denote the computation time of task t_i on processor p_a by W_i^a and the communication delay between t_i and t_k when they are scheduled on processors p_a and p_b by W_{ik}^{ab}. Moreover, the following changes should also be noted.

1. Although we retain most of the terminology, it may have a broader definition than before. For example, we still use the term *processor* to refer to any resource that can execute tasks, but we do not make any assumptions about what it physically represents. Depending on the context, a processor could therefore be anything from a single thread to an entire computing cluster (e.g., in cloud computing).

2. No assumptions are made regarding how related the execution times on different processors are to one another. In our simulations we will treat relatedness as a parameter and consider different cases.

3. We again follow the macro-dataflow model [140] for communication: delays are assumed to be zero between a processor and itself, the network is fully connected and contention for communication resources is not considered. Communication delays are again also assumed to be symmetric, so that $W_{ik}^{ab} = W_{ik}^{ba}$ for all pairs of distinct processors p_a and p_b.

To illustrate the differences between methods for estimating the critical path, throughout this chapter we use a simple example, namely the scheduling of the small graph shown in Figure 3.1 on a target platform comprising two processing resources P1 and P2.

3.2 USES OF THE CRITICAL PATH

The most common way that the concept of the critical path is used in heterogeneous scheduling is to compute priorities for tasks. In particular, tasks which are expected to lie on critical paths are given higher priority than others. Upward ranking from HEFT is the most prominent example: the priority of a task is defined as the estimated critical path length from the task to the sink (inclusive). This approach has proven successful but, as we saw in the previous chapter, the problem is that there are many different ways that the path lengths can be approximated and it typically isn't clear beforehand which will give the smallest schedule makespan for a given DAG and computing platform. Therefore in the following section we will describe multiple methods, before evaluating their performance in HEFT through simulation in Section 3.4.

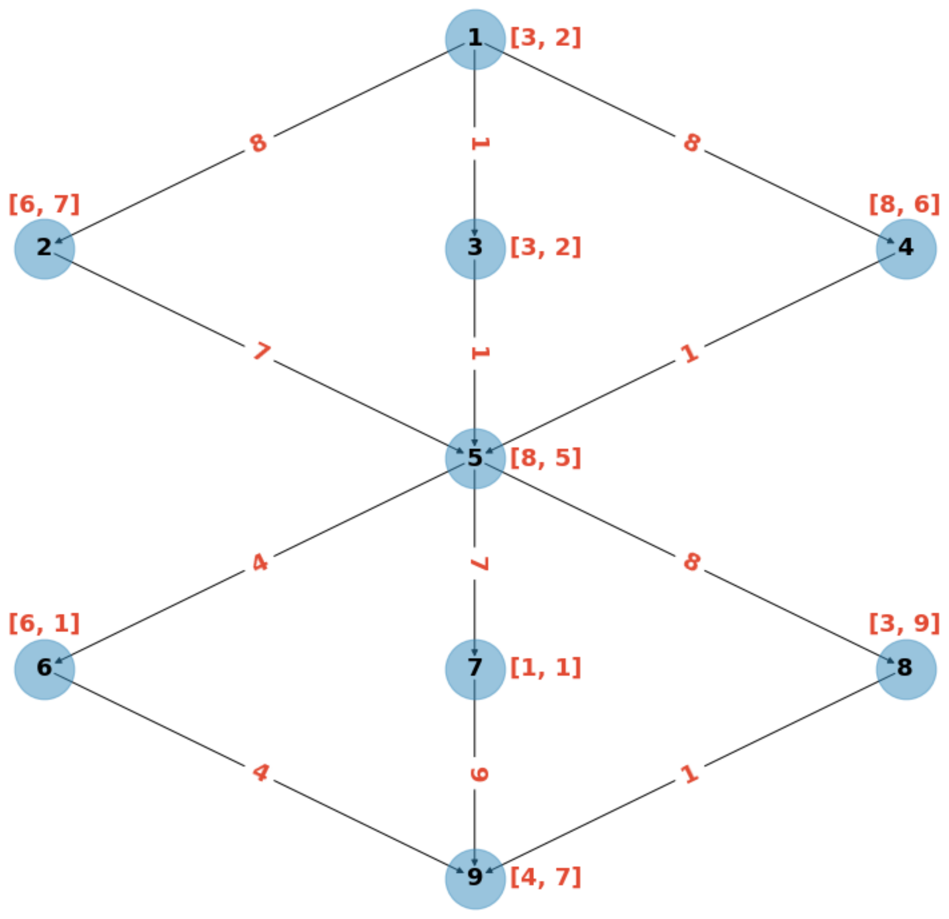


Figure 3.1: Example graph with labels representing computation and (nonzero) communication costs on a two-processor target platform. The bracketed red labels near the vertices represent the task execution times on the two processors in the form $[W_i^1, W_i^2]$ and the edge weights represent the communication cost between tasks when they are scheduled on different processors; note that zero is therefore always an alternative edge weight but is omitted for clarity. Despite its simplicity, there are several different ways that we can define the critical path of this DAG.

Once task priorities have been computed, HEFT schedules them in this order on the processors expected to complete them at the earliest time. But other heterogeneous scheduling heuristics use priorities in different ways. For example, *Hybrid Balanced Minimum Completion Time* (HBMCT) [106] and *Iso-Level Heterogeneous Allocation* (ILHA) [17] both create sets $S_1, S_2, \ldots$ of mutually independent tasks such that the lowest priority task in S_1 has a higher priority than all tasks in S_2, and so on. These sets are then scheduled in order using some heuristic for scheduling independent tasks. In the future it may be worth investigating if the same trends we observe for HEFT also hold for heuristics such as HBMCT and ILHA. However we only use HEFT in our simulations here since it represents arguably the most natural way to use task priorities when scheduling (i.e., schedule the highest priority task on the processor expected to complete it first).

In addition to computing priorities, another use of the critical path is exemplified by the CPOP heuristic. Specifically, all of the tasks along the expected critical path (through the entire DAG, from source to sink) are scheduled on the single processor that minimizes the sum of their execution times; the motivation being to minimize the critical path length by ensuring there are no communication delays along it. Again, the issue here is how we actually determine the path which is most likely to be critical (or which it is most useful to treat as critical). CPOP uses essentially the same method as HEFT to do this, estimating the length of the longest path from each task to the sink (inclusive) in the same way and then adding these to the corresponding estimates from the source to the tasks (exclusive) in order to get estimates of the longest paths from source to sink which pass through each task. The maximal such longest path is then assumed to be critical. Since the idea of treating the critical path separately from the rest of the DAG—which we will refer to as critical path *assignment*—is somewhat different from using critical paths for priorities, in this chapter we will investigate whether the methods for approximating the critical path that perform well in one context also do well in the other.

(Note that CPOP also uses critical path estimates for priorities in the same way as HEFT, with the exception that the priority of a task represents the estimated length of the longest path from source to sink which passes through the task, not just the longest path from that task to the sink. However, when we evaluate different ways to approximate the critical path for assignment we will always use the same task priorities as in the standard HEFT algorithm; the idea being that we want to evaluate which methods for approximating the critical path are most useful in the two different contexts separately. Furthermore, no matter how the critical path is approximated, we consistently found when prioritizing tasks that using the "upward" path length estimate—i.e., from the task to the sink—only was more effective than the sum of the upward and downward path lengths. This makes sense: by the time a task is ready for scheduling, the downward part of the graph has already been scheduled and only the upward critical path estimate path is still relevant.)

3.3 APPROXIMATING THE CRITICAL PATH

Since there is typically no path that will always be critical for any possible labeling of the graph, in practice we need to find some useful approximation instead. Below, we describe multiple different ways this can be done.

3.3.1 Averaging

The most straightforward method is to scalarize the graph weights using some kind of average and then compute path lengths in the usual way for graphs with scalar weights (i.e., dynamic programming). As we saw in the previous chapter, this is generally a useful way to define critical path lengths in priority-based heuristics such as HEFT, which uses arithmetic mean averages by default. The problem is that different average types can lead to different critical paths estimates—and it is very difficult to determine which average will give the smallest makespan for a specific problem beforehand. For example, Figure 3.2 shows how the 14 different averaging schemes defined in Table 3.1—extensions of those from Table 2.2—lead to four possible critical paths for the graph from Figure 3.1. Moreover, as shown in Table 3.2, different averages result in different task priority lists when they are used to compute upward ranks in HEFT—and therefore different schedule makespans. In this case, we see that the W, SW, D and SD averages give the best schedule—which for this small graph it can be verified through exhaustive enumeration is actually the optimal schedule—but it is very difficult to justify any good reason we should anticipate this result.

A comment on terminology: although we refer to the resulting path as *critical*, for many of these averages the motivation is not to identify the path that will be longest. For example, as noted in the previous chapter, the intuition behind the R, D, NC and SD averaging schemes is to find the path with the strongest *preference* for processors of a

Table 3.1: Extensions of the averaging schemes from Table 2.2 for generic heterogeneous scheduling, defined by how they compute the average weight of a task t_i and edge (t_i, t_k).

Average	$\overline{w_i}$	$\overline{w_{ik}}$
M	(2.2)	(2.3)
MD	Median of $\{W_i^a\}_{a=1,\dots,q}$	Median of $\{W_{ik}^{ab}\}_{a,b=1,\dots,q}$
B	$\min_a W_i^a$	0, if $r_i := \arg\min_a W_i^a = r_k$, else $W_{ik}^{r_i r_k}$
SB	$\min_a W_i^a$	0
W	$\max_a W_i^a$	0, if $r_i := \arg\min_a W_i^a = r_k$, else $W_{ik}^{r_i r_k}$
SW	$\max_a W_i^a$	$\max\{W_{ik}^{ab}\}_{a,b=1,\dots,q}$
HM	(2.15)	(2.16)
SHM	(2.15)	0
GM	Geometric mean of $\{W_i^a\}_{a=1,\dots,q}$	Geometric mean of $\{W_{ik}^{ab}+1\}_{a,b=1,\dots,q}$
SGM	Geometric mean of $\{W_i^a\}_{a=1,\dots,q}$	0
R	$\max_a W_i^a / \min_a W_i^a$	0
D	$\max_a W_i^a - \min_a W_i^a$	$\max\{W_{ik}^{ab}\}_{a,b=1,\dots,q}$
NC	D/R	0
SD	Std. dev. of $\{W_i^a\}_{a=1,\dots,q}$	Std. dev. of $\{W_{ik}^{ab}\}_{a,b=1,\dots,q}$

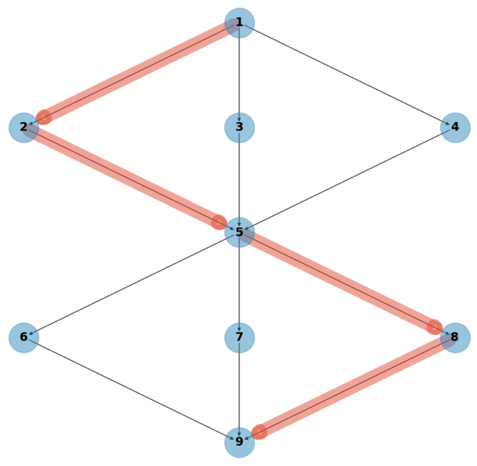
certain type. However, we will continue to refer to the computed paths as critical since the underlying motivation is always that the identified path is in some sense the most important.

3.3.2 Bounds

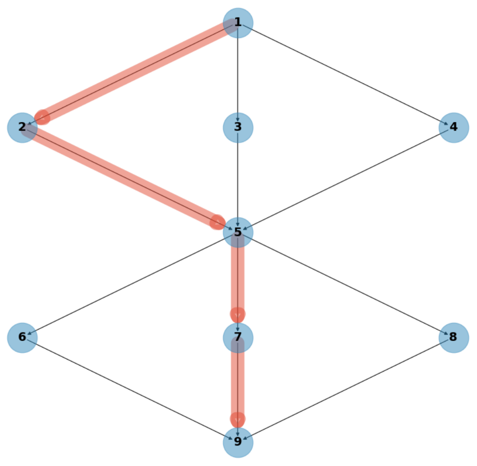
It is interesting to note that, for many average types, the expected critical path length may be greater than the optimal schedule makespan. For example, using the M average for the graph from Figure 3.1 gives an expected critical path length of 39 (compared to an

Table 3.2: Task priority lists corresponding to different averaging schemes and the resulting HEFT schedule makespans for the graph from Figure 3.1. In case of identical priorities, the task with the higher numerical index was given higher priority.

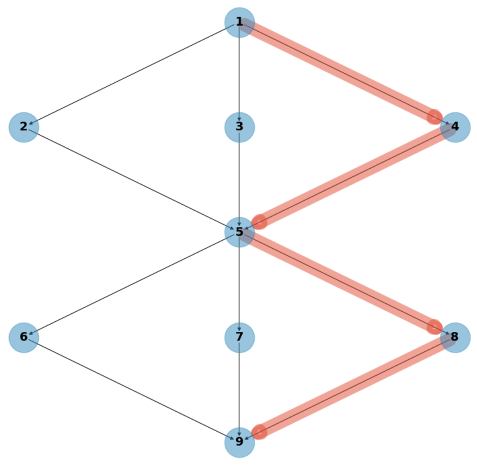
Priority list	Averages	Makespan
(1, 2, 4, 3, 5, 8, 6, 7, 9)	M, MD, SB, GM	41
(1, 2, 4, 3, 5, 6, 8, 7, 9)	B	41
(1, 2, 4, 3, 5, 6, 7, 8, 9)	W, SW, D	35
(1, 2, 4, 3, 5, 8, 7, 6, 9)	HM	38
(1, 4, 2, 3, 5, 8, 6, 7, 9)	SHM, SGM	41
(1, 3, 4, 2, 5, 6, 8, 7, 9)	R	37
(1, 4, 2, 3, 5, 8, 6, 9, 7)	NC	41
(1, 2, 4, 3, 5, 7, 6, 8, 9)	SD	35



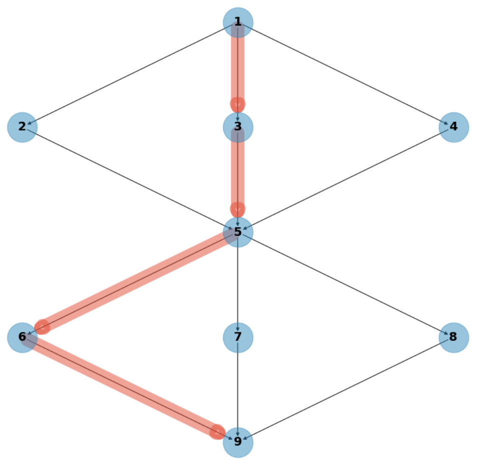
(a) M, MD, B, SB, W, SW, GM.



(b) HM, D, SD.



(c) SHM, SGM, NC.



(d) R.

Figure 3.2: Critical paths for the graph from Figure 3.1 estimated using the indicated averaging schemes. We see that different averages can lead to very different critical paths.

optimal schedule makespan of 35). This is not truly problematic—as noted above, many of the average types do not even attempt to achieve this—but it is notable in light of the fact that the critical path length usually represents a lower bound on the makespan, so that minimizing the former gives the most scope to minimize the latter. It may therefore be useful to compute a lower bound on the critical path length, since this would also be a lower bound on the makespan of *any* schedule.

The most straightforward approach is to just set all of the weights to their minimal values, as in the SB averaging scheme, and then compute the longest paths through the resulting scalar-weight graph. For the example, this gave a lower bound of 20, as compared to the optimal makespan of 35. The problem with this method is that it is too optimistic: edge weights are always set to their smallest values, even if this is impossible given the weights assigned to the tasks connected by the edge. The B average attempts to compensate for this but is too restricted by assuming that all tasks are scheduled on their fastest processor; in the example, the corresponding critical path length was 43, which is actually greater than the optimal makespan.

We can however easily find a tighter bound on the critical path length using recursion. In particular, let $\ell_n^a = W_n^a$ for all $a = 1, \ldots, q$, then work up through the task graph and compute

$$\ell_i^a = W_i^a + \max_{k \in \Gamma_i^+} \Big(\min_{b=1,\ldots,q} \{ W_{ik}^{ab} + \ell_k^b \} \Big) \tag{3.1}$$

for all $i = n-1, \ldots, 1$. Intuitively, ℓ_i^a represents the shortest possible critical path that can be achieved assuming that task t_i is scheduled on processor p_a. A lower bound on the critical path length from task t_i to the sink is then given by $\ell_i := \min_a \ell_i^a$, so that in particular ℓ_1 is a bound on the critical path through the entire DAG—and therefore the optimal makespan. For the graph from Figure 3.1, we find that we have $\ell_1 = 28$, which is a tighter bound than that obtained through the SB averaging scheme. Moreover, defining the priorities of all tasks in HEFT as their ℓ values results in a schedule with makespan 35—i.e., an optimal one.

Note that the ℓ_i^a are essentially just the *optimistic costs* from the PEFT heuristic [12], although, as discussed in Section 2.3.2, the actual communication delay W_{ik}^{ab} is used in the minimization rather than an average value since that is the only way to get a true lower bound. Moreover, as also stated in the previous chapter, the cost of computing the ℓ_i^a is only $O(n^2)$—i.e., the same complexity in n as the averaging method.

If we need to identify an actual path for assignment which corresponds to the critical path lower bound, this can be done in a straightforward manner by working forward through the DAG after the ℓ values have been computed and successively determining which child task gives the desired path length. Alternatively, we could simply keep track of the maximizing child task when computing the values using Eq. (3.1). At any rate, for the example we find that the path corresponding to the lower bound is $(1, 2, 5, 8, 9)$, which in this case is actually the same identified by the B/SB averaging schemes, as shown in Figure 3.2.

Just as the method described above extends the motivation behind the B/SB averages, we could also extend the W/SW averages and find *upper* bounds on the critical path lengths. This can be done by simply replacing the inner minimization in (3.1) with a maximization over the processors instead—i.e., recursively computing

$$z_i^a = W_i^a + \max_{k\in\Gamma_i^+}\Big(\max_{b=1,\dots,q}\{W_{ik}^{ab} + z_k^b\}\Big) \tag{3.2}$$

for all tasks t_i and processors p_a, and then defining $z_i = \max_a z_i^a$. For the example, this gives an upper bound on the critical path through the entire DAG of $z_1 = 57$, which is the same as that obtained by the W averaging scheme (and less than that of SW, which therefore represents an infeasible task assignment). It is difficult to intuitively justify why we might prefer to use an upper bound on the critical path length since, unlike a lower bound, it does not induce a corresponding bound on the schedule makespan. However, given the strong performance of the W/SW averages in the previous chapter it seems sensible to at least consider it experimentally.

3.3.3 Stochastic interpretation

By default HEFT uses arithmetic mean averages to scalarize the weights of the DAG and approximate the critical path. One way to interpret this approach is that since the actual values the weights will take at runtime are unknown, we are essentially quantifying the likelihood that they will take certain values. In other words, the actual weight that a task t_i takes at runtime is treated as a discrete *random variable* (RV) w_i which takes values from the set $\{W_i^a\}_{a=1,\dots,q}$ according to some approximated *probability mass function* (pmf)—and similarly for the actual weight $w_{ik} \in \{W_{ik}^{ab}\}_{a,b=1,\dots,q}$ of a generic edge (t_i, t_k). Using arithmetic mean values implies that all weights are equally likely, so that the pmf of w_i is defined by

$$m_i(W_i^a) := \mathbb{P}[w_i = W_i^a] = \frac{1}{q}, \quad \forall a = 1, \dots, q, \tag{3.3}$$

and the pmf of w_{ik} by

$$m_{ik}(W_{ik}^{ab}) := \mathbb{P}[w_{ik} = W_{ik}^{ab}] = \frac{1}{q^2}, \quad \forall a, b = 1, \dots, q. \tag{3.4}$$

Note that in this case the expected values of the task and edge weight variables are therefore given by

$$\mathbb{E}[w_i] = \sum_{a=1}^{q} W_i^a m_i(W_i^a) = \frac{1}{q} \sum_{a=1}^{q} W_i^a = \overline{w_i},$$
$$\mathbb{E}[w_{ik}] = \sum_{a=1}^{q} \sum_{b=1}^{q} W_{ik}^{ab} m_{ik}(W_{ik}^{ab}) = \frac{1}{q^2} \sum_{a,b} W_{ik}^{ab} = \overline{w_{ik}},$$

where $\overline{w_i}$ and $\overline{w_{ik}}$ are as defined by Eqns. (2.2) and (2.3), respectively. This means that setting all DAG weights to arithmetic mean values and then computing upward ranks of the corresponding graph with scalar weights is equivalent to setting $u_n = \mathbb{E}[w_n]$, then moving up the DAG and recursively computing

$$u_i = \mathbb{E}[w_i] + \max_{k \in \Gamma_i^+} \left(u_k + \mathbb{E}[w_{ik}] \right) \tag{3.5}$$

for all other tasks. Let G_s be the counterpart of the graph G with stochastic task and edge weights. Clearly, if the weights of G_s are RVs, then the length of the critical path between a task t_i and the sink, which we will denote by L_i, must itself be an RV. The question that arises is: what is the relationship between u_i and L_i? In fact, it has long been known that u_i is a *lower bound* on the expected value of L_i—i.e., we have $u_i \leq \mathbb{E}[L_i]$ for all $i = 1, \dots, n$ [55], [79]. This prompts a follow-up question: could L_i, or its expected value, be more useful than u_i?

The Monte Carlo method. Unfortunately, computing the distribution of the longest path through a DAG with stochastic weights, or even just its expected value, is a very difficult problem, as we will see in Chapter 4, which is devoted to this topic. However, it suffices to say here that *Monte Carlo* (MC) *simulation* is currently the gold standard method for approximating the longest path distribution—at least when the computational cost is ignored and the weight distributions are known. In this context, MC simulation refers to realizing the weights of the DAG according to their pmfs and computing the longest path of the resulting scalar-weighted graph. This is done repeatedly, giving a set of longest path instances whose empirical distribution function is a good approximation of the true distribution function, assuming that we take adequately many realizations.

The downside of the MC method is that it may be expensive: we need to repeatedly realize all $n + v \approx n^2$ weights of the DAG and compute the longest path each time, itself an n^2 operation. (This is why we focus in the next chapter largely on devising cheaper alternatives!) Assuming that we take R realizations of the graph weights, we would perhaps expect the MC method to take R times longer than computing upward ranks for averaged weights since in that case we only need to compute the longest path through a single graph with fixed weights. In our own implementations, however, we found that the ratio between the two runtimes was rarely so extreme because the MC method can largely be vectorized; this will be discussed at greater length in the next chapter. At any rate, although additional comments will be made about runtimes later, we are more interested here in establishing whether tightening the bound on $\mathbb{E}[L_i]$ is useful at all.

Using the data. Through the MC method, we can generate many different realizations of the task graph. For each task we therefore have a set of realized critical path lengths to the sink. For a generic task t_i, assume that the relevant path lengths have sample mean μ_i and sample standard deviation σ_i. The question is, what is the best way to use this data to compute a priority for t_i? The obvious choice is to define the priority as the sample mean of the relevant path length data. Since $\mu_i \approx \mathbb{E}[L_i]$, this is the natural extension of the upward ranking. However, as noted by Van Slyke [133], when prioritizing two or more tasks it is arguably most sensible to compare their *criticalities*, the probability that each will lie on a critical path (through the entire DAG). This seems reasonable, although, as elucidated by Williams [139], the underlying intuition can be deceptive. At any rate, since we can easily approximate the criticality of a task by dividing the number of times it lay on a path that was observed to be critical during the MC by the total number of realizations, we consider this alternative in our experimental comparison later.

(Note that defining task priorities as their estimated criticality does not guarantee that a parent task's priority is always greater than a child's—i.e., the precedence constraints could be violated. However, as in the previous chapter, this can be remedied by simply selecting the task with the highest priority from those that are currently ready for scheduling.)

As for tasks, we can define the criticality of a *path* as the probability that it will become critical—i.e., longer than all other paths [84]. If we wish to identify a complete path through the DAG for assignment, as in CPOP, the path with the highest criticality would appear to be the natural choice. Identifying such a path analytically is a difficult problem, as we will see in the next chapter, but we can simply approximate it by using the path which was most frequently observed to be critical for the realized graphs during the MC. Therefore we will also evaluate the utility of this MC-based method for approximating the critical path.

Harmonic pmfs. Thus far we have implicitly assumed that the pmfs of the task and edge weights reflect arithmetic mean averages. As noted in Section 2.4.1, one issue with this approach is that all processor selections are regarded as equally likely, whereas we would expect that tasks are more likely to be scheduled on processors with smaller execution times. To remedy this, we could define the weight pmfs to represent the harmonic mean (HM) averaging scheme instead. In particular, let

$$h_i = \sum_{a=1}^{q} \frac{1}{W_i^a}$$

and define an alternative set of pmfs by

$$\hat{m}_i(W_i^a) = \frac{1}{W_i^a h_i}, \quad \forall i = 1, \dots, n \text{ and } a = 1, \dots q, \tag{3.6}$$

and

$$\hat{m}_{ik}(W_{ik}^{ab}) = \hat{m}_i(W_i^a) \cdot \hat{m}_k(W_k^b) = \frac{1}{W_i^a W_k^b h_i h_k}, \quad \forall i, k, a, b. \tag{3.7}$$

Example. Since we described four different ways to compute task priorities based on the stochastic interpretation of the problem outlined above, we consider the simple graph from Figure 3.1 in order to illustrate how they work. The four methods are each referred to by a string X-Y, where X indicates how the MC data is used to compute task priorities and Y which pmfs were used when realizing the weights. The two choices for X are:

- EV (for *expected value*), which indicates that we define task priorities as the sample means of their realized critical path lengths (from the task to the sink);
- CR (for *criticality*), which represents using the observed criticality of tasks for priorities.

Table 3.3: HEFT schedule makespans corresponding to different methods of using MC data to compute priorities for the graph from Figure 3.1. In case of ties, the task with the higher numerical index was given higher priority.

Method	Priority list	Makespan
EV-A	(1, 2, 4, 3, 5, 8, 7, 6, 9)	38
EV-H	(1, 2, 4, 3, 5, 7, 6, 8, 9)	35
CR-A	(1, 2, 4, 3, 5, 8, 7, 6, 9)	38
CR-H	(1, 2, 4, 3, 5, 8, 7, 6, 9)	38

For Y, the two options are A (arithmetic), which denotes using pmfs (3.3) and (3.4) for the MC sampling, or H (harmonic) which indicates that the pmfs (3.6) and (3.7) are used instead. In both this example and the investigation detailed in Section 3.4, we always did 1000 MC realizations for both pmf choices. As we will see in the next chapter, this is usually adequate to obtain highly accurate estimates of the expected value of the critical path length. Comments will be made later about the time complexity implications of this choice. Table 3.3 presents the task priority lists, and HEFT schedule makespans obtained by using those lists, for the four methods. We see that EV-H successfully returns the optimal schedule and all the others at least better the makespan of the standard HEFT algorithm (41).

With regard to what we have dubbed critical path assignment, we also recorded which paths (through the entire DAG) were critical and how often this the case. Table 3.4 contains all the paths which were observed to be critical for 1000 realizations of the graph, with both A and H pmfs, and how often this was the case. We see that three of the possible critical paths from Figure 3.2 are present, in addition to three others. Since there are only nine different paths through this graph, two-thirds of them were therefore critical at least once. For both choices of pmfs, path (1, 2, 5, 8, 9) was most frequently critical—i.e., we assume it has the greatest criticality. However, it was only actually critical about 30% of the time—in other words, there was a 70% probability that it was not critical for any single realization of the graph. This perhaps suggests that critical path assignment is not well-suited for this example.

Table 3.4: Paths which were observed to be critical, and how frequently, for 1000 MC realizations of the graph from Figure 3.1.

Path	PMF = A	PMF = H
(1, 2, 5, 6, 9)	137	84
(1, 2, 5, 7, 9)	193	256
(1, 2, 5, 8, 9)	290	302
(1, 4, 5, 6, 9)	70	34
(1, 4, 5, 7, 9)	133	164
(1, 4, 5, 8, 9)	177	160

3.4 SIMULATION RESULTS

To gauge how the different methods of defining critical paths perform empirically, we used a custom software simulator, the source code for which can be found at the Github repository for this thesis[1]. This software is very similar to that used in the previous chapter, with the exception of the changes stated in Section 3.1—most notably, the fact that arbitrarily many different processors are now possible, not just two different types.

3.4.1 Graphs

As before, we used two sets of task graphs in this experimental investigation, one based on Cholesky factorization and the other based on randomly generated topologies taken from the STG [128]. However, the manner in which we generated computation and communication costs differs from the previous chapter since we now wish to consider arbitrary heterogeneous computing environments. Broadly speaking, generating useful artificial costs for scheduling simulations is a difficult problem and many different methods have been proposed in the literature; a good overview can be found at [32]. The methods that we decided to use are described below, but it should be emphasized that there are alternatives that could have been used instead.

Cholesky DAGs. Once again we constructed ten different topologies corresponding to the Cholesky factorization of an $N \times N$ tiled matrix, for $N = 5, 10, \dots, 50$. To set the weights of these graphs, we broadly followed the *noise-based* algorithm described in [32], with some modifications due to the specific structure of Cholesky factorization DAGs and

[1]https://github.com/mcsweeney90/thesis-code

the additional need to generate communication costs. The method works by sampling costs randomly from gamma distributions. Expected values of these distributions are typically—but not always—assumed here to be one, so that they are specified by their coefficients of variation (the ratio of the standard deviation and the expected value). The motivation behind this approach is that small coefficients of variation typically represent low heterogeneity in the costs, whereas larger values indicate greater heterogeneity. Note that other distributions could be used instead, but gamma distributions are easy to work with and are often used for modeling task execution times in heterogeneous scheduling [19], [32].

The cost-setting algorithm proceeds as follows. First, each of the q processors is assigned a "speed" by sampling randomly from a gamma distribution with unit mean and coefficient of variation v_{proc}. Denote the speed of processor p_a by S_a. Next, calculate the "size" of each task by assuming that these correspond to the number of *flops*—floating point operations—required for the kernel type. Once a tile size nb has been specified, these are well-known: `GEMM` tasks require $2nb^3$ flops, `POTRF` $nb^3/3 + nb^2/2 + nb/6$, `SYRK` $nb^2(nb+1)$, and `TRSM` nb^3 [103]. Rather than requiring an input tile size, we normalize the task sizes according to the constant multiplying the nb^3 term, so that `GEMM` is assigned a relative size 2, `POTRF` 1/3, `SYRK` 2, and `TRSM` 1. For all $i = 1, \ldots, n$, let T_i denote the kernel type of task t_i, and define a function $F(T_i)$ which returns the relative size of kernels of type T_i, so that, for example, $F(T_i) = 1/3$ if $T_i =$ `POTRF`. Under a fully related scheduling model, the computation time of task t_i on processor p_a would then simply be given by $W_i^a = F(T_i)/S_a$. However, we instead define

$$W_i^a = R(T_i) \times F(i)/S_a,$$

where, for each kernel type k, $R(k)$ is sampled from a gamma distribution with unit mean and coefficient of variation v_{rel}, $R(k) \sim \Gamma(1/v_{\text{rel}}^2, v_{\text{rel}}^2)$. The idea here that is the $R(k)$ value roughly characterizes how related the computation times are for that kernel; if $R(k) = 1$, then they are perfectly related, whereas much larger or smaller values indicate wider variability. Note therefore that large values of v_{rel} correspond to less related computation costs than smaller ones.

Generating communication costs is somewhat simplified since we assume that all tile sizes are the same and therefore each task transmits the same amount of data. This means that we effectively only have to construct a single symmetric $q \times q$ matrix C of communication costs such that $C(a,b) = W_{ik}^{ab}$ for any edge (t_i, t_k) and all $a, b = 1, \ldots, q$. Recall that the communication cost from any processor to itself is zero, and that all costs are assumed to be the same in both directions for each processor pair. Therefore all diagonal entries of C are zero and it is symmetric, so we only need to populate the upper (resp. lower) triangular part. Again, we use a gamma distribution for this, but this time its mean value μ_{comm} must first be calculated in order for the DAG to meet some input target CCR parameter β, where β is as defined by Eq. (2.22). Let $\overline{C}$ be the mean of all entries in C, including the zeroes along the diagonal. Note that $\mu_{\text{comm}} = \frac{q}{q-1}\overline{C}$ because there are q zeroes along the diagonal. Since all edges have the same possible costs, we have $\overline{w_{ik}} = \overline{C}$ for any (t_i, t_k). This means in particular that

$$\beta = \frac{\sum_{i,k} \overline{w_{ik}}}{\sum_i \overline{w_i}} = \frac{v\overline{C}}{\sum_i \overline{w_i}} \implies \overline{C} = \frac{\beta}{v} \sum_i \overline{w_i}.$$

So, once the computation costs have been set, we can calculate $\overline{C}$ and therefore μ_{comm}. As before, the distribution is also specified by an input coefficient of variation v_{band}, this time intended to represent the variability of the link bandwidths, so that for all $a = 1, \ldots, q$ and $b = a + 1, \ldots, q$, we sample

$$C(a,b) \sim \Gamma(1/v_{\text{band}}^2, \mu_{\text{comm}} \cdot v_{\text{band}}^2)$$

in order to construct the matrix.

To summarize, the complete cost-setting procedure we used for the Cholesky graphs is presented in Algorithm 3.1. In principle there are five parameters which control the method: q, β, v_{proc}, v_{rel} and v_{band}. However to simplify the presentation of results later we assume that $v_{\text{proc}} = v_{\text{rel}} = v_{\text{band}} := V$ so that there is effectively only one coefficient of variation parameter. In our experiments we considered $V \in \{0.2, 1.0\}$ in order to compare a relatively low variation in the costs to much greater variability. Furthermore, we assumed that the target platform always comprised $q = 20$ processors since the different sizes of the graphs ensured that the ratio between tasks and processors varied widely anyway. For the final parameter, β, we considered values from the set $\{0.01, 0.1, 1.0, 10.0\}$. For each of the 10 topologies and each combination of the parameters V and β, we did 10 realizations of the costs. Altogether this means that the Cholesky DAG set effectively comprised $10 \times 2 \times 4 \times 10 = 800$ different graphs.

Algorithm 3.1: Cost-setting procedure for Cholesky DAGs.

```
   Inputs: $q, \beta, v_{\text{proc}}, v_{\text{rel}}, v_{\text{band}}$
   // Set relative sizes of task types
1  $F(\texttt{GEMM}) = 2$, $F(\texttt{POTRF}) = 1/3$, $F(\texttt{SYRK}) = 2$, $F(\texttt{TRSM}) = 1$
   // Sample processor speeds
2  for $a = 1, \dots, q$ do
3      $S_a \sim \Gamma(1/v_{\text{proc}}^2, v_{\text{proc}}^2)$
4  end
   // Sample relatedness multipliers
5  for $k \in \{\texttt{GEMM}, \texttt{POTRF}, \texttt{SYRK}, \texttt{TRSM}\}$ do
6      $R(k) \sim \Gamma(1/v_{\text{rel}}^2, v_{\text{rel}}^2)$
7  end
   // Set computation costs
8  for $i = 1, \dots, n$ do
9      for $a = 1, \dots, q$ do
10         $W_i^a = R(T_i) \times F(T_i)/S_a$
11     end
12 end
   // Calculate distribution mean for communication costs
13 $\mu_{\text{comm}} = \frac{q\beta}{v(q-1)} \sum_i \overline{w_i}$
   // Construct communication cost matrix
14 for $a = 1, \dots, q$ do
15     for $b = a+1, \dots, q$ do
16         $C(a, b) \sim \Gamma(1/v_{\text{band}}^2, \mu_{\text{comm}} \cdot v_{\text{band}}^2)$
17     end
18 end
```

Randomly generated DAGs. We took a slightly different approach for the randomly generated graphs from the STG. First, to generate the computation costs, we used the *correlation noise-based* (CNB) method introduced in [32]. This is an extension of the standard method, based on a new measure called the *correlation* that more rigorously quantifies the "relatedness" of a set of costs. In addition to the number of processors q, the CNB method takes three parameters, r_{task}, r_{proc} and V, where:

- $r_{\text{task}} \in [0, 1)$ roughly describes how related the task sizes are (with low values being less related);
- $r_{\text{proc}} \in [0, 1)$ does likewise for the processor speeds;
- V is the coefficient of variation for the (gamma) distributions that the costs are sampled from.

(There is also a fourth parameter, μ, the mean of the sampling distributions for the costs, however for simplicity we always assumed that $\mu = 1$ so it will not be mentioned again.) A full description of the CNB method can be found at [32], and the values of the parameters that we used are stated below.

After setting the computation costs with the CNB method, we used a different method to generate communication costs than for the Cholesky graphs since we no longer assume that all of the tasks transmit the same amount of data. First, we randomly generate the bandwidth of all links by sampling from a gamma distribution with unit mean and coefficient of variation v_{band}—i.e.,

$$B(a, b) \sim \Gamma(1/v_{\text{band}}^2, v_{\text{band}}^2),$$

where $B(a, b) = B(b, a)$ is the bandwidth between any pair of processors p_a and p_b such that $b > a$. Then we compute

$$\overline{B} = \sum_{a=1}^{q} \sum_{b=a+1}^{q} \frac{1}{B(a, b)},$$

as this will be needed later. Next, for each task t_i, we sample what we call the *local CCR* β_i from a $\Gamma(1/v_{\text{ccr}}^2, \mu_{\text{ccr}} \cdot v_{\text{ccr}}^2)$ distribution, where μ_{ccr} and v_{ccr} are parameters. The local CCR of a task t_i is defined as the ratio of the task's mean communication and computation times, $\beta_i = \overline{w_{ik}}/\overline{w_i}$ (for any $k \in \Gamma_i^+$ since we assume that tasks transmit the same amount of data to all their children). The CCR as defined by Eq. (2.22) is therefore the mean local

CCR of all tasks multiplied by v/n (where v is the number of edges). Once the local CCR of a task has been sampled, we calculate the corresponding amount of data d_i that the task needs to transmit in order to achieve the specified local CCR through

$$d_i = \frac{\overline{w_i}\beta_i q^2}{2\overline{B}}. \tag{3.8}$$

Finally, for all $k \in \Gamma_i^+$, $a = 1, \dots, q$, and $b = a+1, \dots, q$, we set

$$W_{ik}^{ab} = W_{ik}^{ba} = d_i / B(a,b).$$

Algorithm 3.2 summarizes the complete procedure used to set the communication costs for the DAGs in the STG set. Note that we assumed that the same coefficient of variation V used for setting the computation costs is also used for the local CCR and bandwidth coefficients of variation—i.e., $v_{\text{ccr}} = v_{\text{band}} = V$. As for the Cholesky factorization graphs, this means that in effect we had only one coefficient of variation parameter for the entire cost-setting method.

Algorithm 3.2: Procedure used to set communication costs for STG set.

```
   Inputs: $q, \mu_{\text{ccr}}, v_{\text{ccr}}, v_{\text{band}}$
   // Set bandwidths
 1 for $a = 1, \dots, q$ do
 2     for $b = a+1, \dots, q$ do
 3         $B(a,b) \sim \Gamma(1/v_{\text{band}}^2, v_{\text{band}}^2)$
 4     end
 5 end
 6 for $i = 1, \dots, n$ do
 7     $\beta_i \sim \Gamma(1/v_{\text{ccr}}^2, \mu_{\text{ccr}} \cdot v_{\text{ccr}}^2)$
 8     Compute $d_i$ using Eq. (3.8)
 9     for $k \in \Gamma_i^+$ do
10         for $a = 1, \dots, q$ do
11             for $b = a+1, \dots, q$ do
12                 $W_{ik}^{ab} = d_i / B(a,b)$
13             end
14         end
15     end
16 end
```

Recall that the STG benchmark comprises multiple subsets, each with 180 DAGs of a given size. We used only those graphs with $n = 100$ tasks since the effect of varying DAG size should be easier to ascertain from the Cholesky set. Including both the CNB method for setting computation costs and Algorithm 3.2 for the communication costs, the parameters which control the complete cost-setting procedure, and the values that we considered, are as follows:

- $q \in \{2,4,8\}$;
- $r_{\text{task}} \in \{0.1, 0.5, 0.9\}$;
- $r_{\text{proc}} \in \{0.1, 0.5, 0.9\}$;
- $V \in \{0.2, 1.0\}$;
- $\mu_{\text{ccr}} \in \{0.01, 0.1, 1.0, 2.0\}$.

For each graph topology and combination of parameters, we repeated the algorithm 3 times, so that altogether the STG set effectively comprised $180 \times 3 \times 3 \times 3 \times 3 \times 2 \times 4 = 116640$ different graphs. Note that we considered $\mu_{\text{ccr}} = 2.0$ rather than 10.0 because we found, as in the previous chapter, that this led to an unacceptably high number of *failures*—i.e., a worse schedule than the minimal serial time—for all of the methods compared. Even for $\mu_{\text{ccr}} = 2.0$, the failure rates were typically around 5%, which is perhaps still too high, but this rose to 50% for $\mu_{\text{ccr}} = 10.0$ which is clearly unacceptable. High failure rates were also apparent for small Cholesky graphs with $\beta = 10$ but we still present that data because they were much rarer for larger ones.

3.4.2 Task priorities

We evaluated the following methods for computing critical path lengths as task priorities in HEFT.

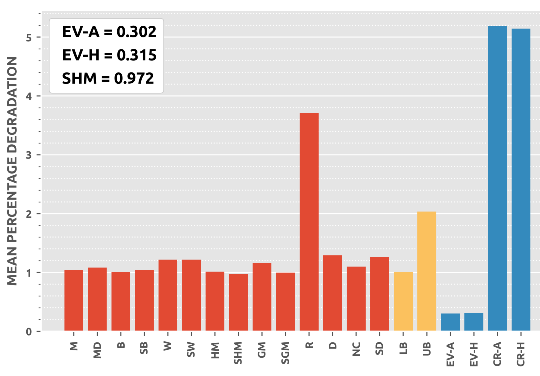
- The 14 averaging schemes defined in Table 3.1;
- The lower and upper bounds on the critical path lengths defined in Section 3.3.2 (which will be referred to as LB and UB, respectively);
- The 4 Monte Carlo-based methods EV-A, EV-H, CR-A and CR-H that were defined in Section 3.3.3. (Note that, as in the example, we always used 1000 realizations of the DAG weights in the MC simulation.)

Altogether, this means that there were 20 different methods that we compared to one another.

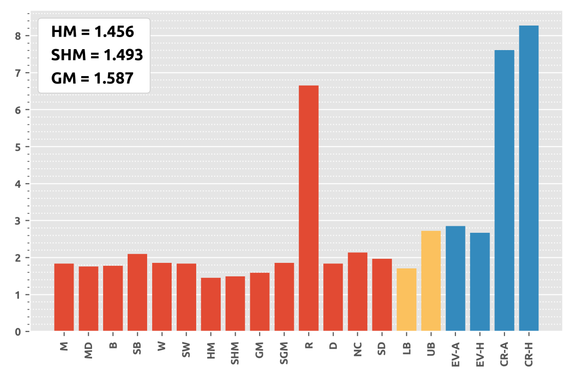
Cholesky DAGs. In Figure 3.3, we illustrate the mean percentage degradation (MPD) of each task prioritization scheme for the Cholesky factorization graphs. Note that the figure divides the graphs according to V and β, so that each bar chart presents the results for 100 (differently sized) graphs. As in the previous chapter, we see that there is a lot of variation and no method completely dominates all others. The new EV-A/H schemes were clearly the best for $V = 0.2$, especially for lower β values, although it should be noted that—as the MPD values stated in the figure indicate—the margins were very fine for those graphs, with almost all of the methods doing well. There was little advantage in using EV-H rather than EV-A and they were usually very similar. Results were much less positive for CR-A/H since they were the worst methods overall. Falling between the two extremes, the bound-based task prioritizations LB and UB were both decidedly mediocre, although the former was consistently superior to the latter. As for the averages, the means M, HM/SHM and GM/SGM all performed well, especially GM, which recorded the lowest MPD for the set as a whole.

STG set. We repeated the comparison for the STG set, as summarized by Figure 3.4. This time we present the data according to the values of V and μ_{ccr} that were used in the cost-setting method, so that each bar chart represents $116640/8 = 14580$ graphs. Overall, the comparative quality of the different prioritization methods was largely similar to what we observed for the Cholesky graphs. However the two EV variants were clearly the outright best overall this time, almost always being the top two methods for each subset of the graphs shown in the figure. EV-H was also slightly, but consistently, superior to EV-A, where we saw no difference before. On the other hand, once again the criticality-based schemes CR-A and CR-H were the worst and the bounds LB and UB were mediocre across the board. Furthermore, the same averages that did well for the Cholesky graphs—i.e., M, HM/SHM, GM/SGM—also did well for these graphs. Note that we chose to illustrate the results in Figure 3.4 according to V and μ_{ccr} since the comparative performance of the task prioritizations was more consistent for different values of the other parameters.

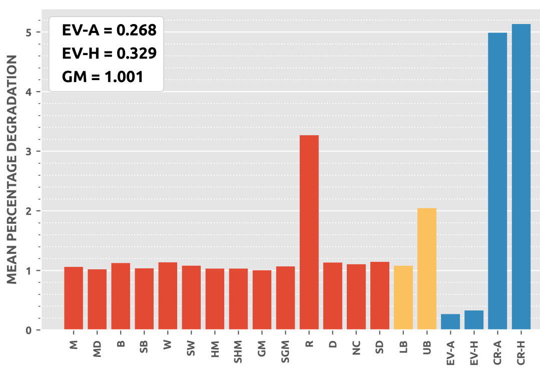
Despite the good performance of the two EV methods, it is still unclear that they offer any real advantage once the additional computational effort required to compute them is taken into account. We are wary of discussing runtimes here since our `Python` code is not—and is not intended to be—optimal, but we found that even EV-A (which was



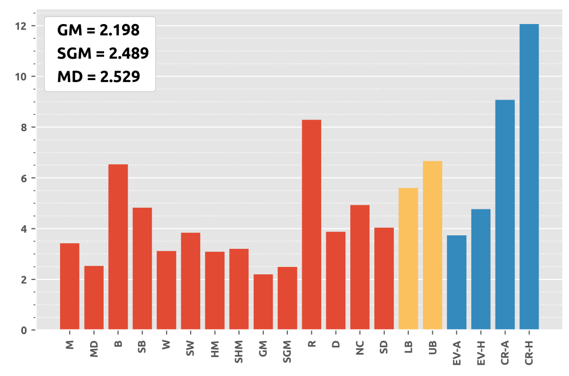
(a) $V = 0.2$, $\beta = 0.01$.



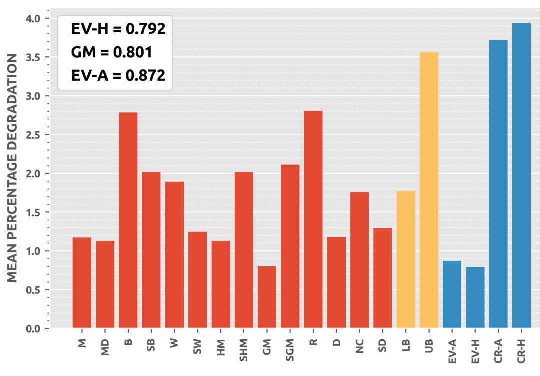
(b) $V = 1.0$, $\beta = 0.01$.



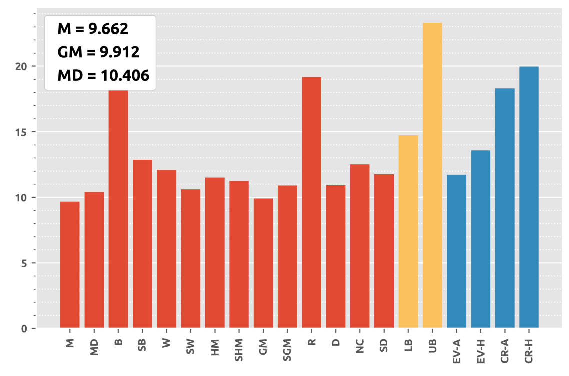
(c) $V = 0.2$, $\beta = 0.1$.



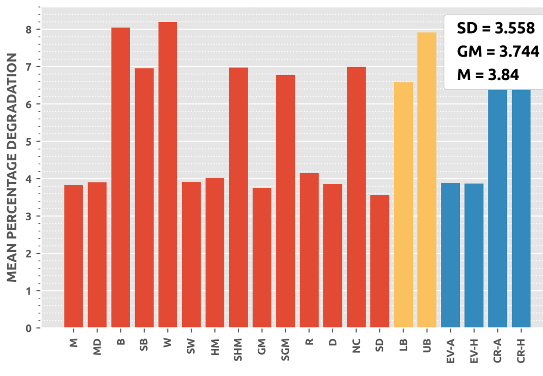
(d) $V = 1.0$, $\beta = 0.1$.



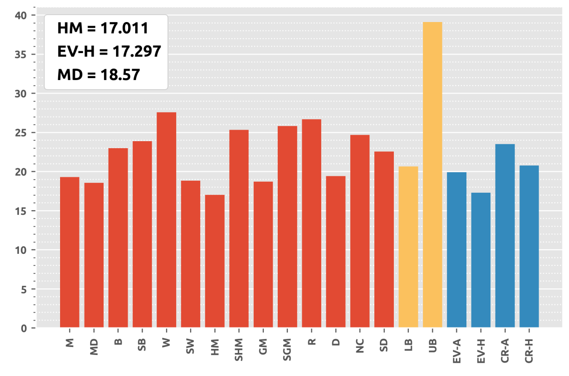
(e) $V = 0.2$, $\beta = 1.0$.



(f) $V = 1.0$, $\beta = 1.0$.

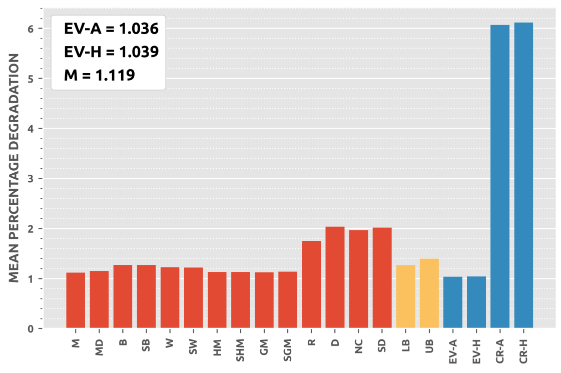


(g) $V = 0.2$, $\beta = 10.0$.

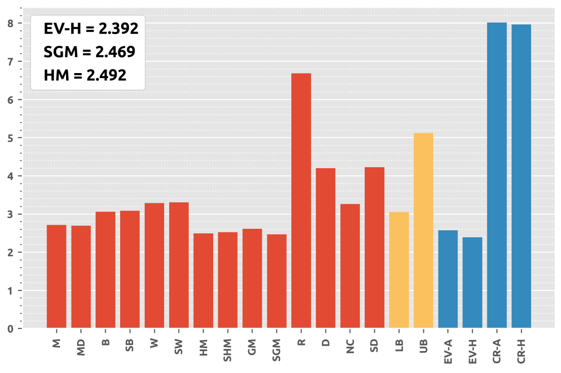


(h) $V = 1.0$, $\beta = 10.0$.

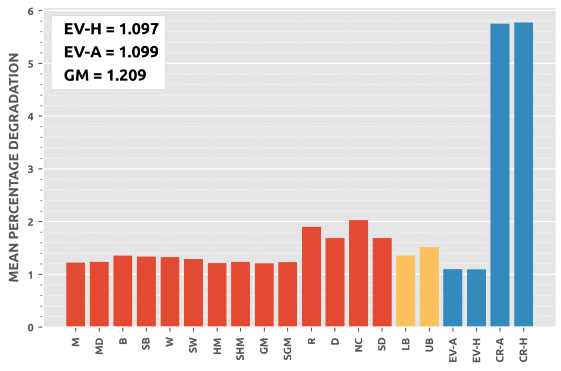
Figure 3.3: Mean percentage degradation (MPD) of task prioritization schemes for Cholesky set with different combinations of V (coefficient of variation) and β (the CCR). Legends identify the three best. Red bars represent averaging-based schemes, yellow those based on bounds on the critical path length, and blue those derived from the stochastic interpretation described in Section 3.3.3.



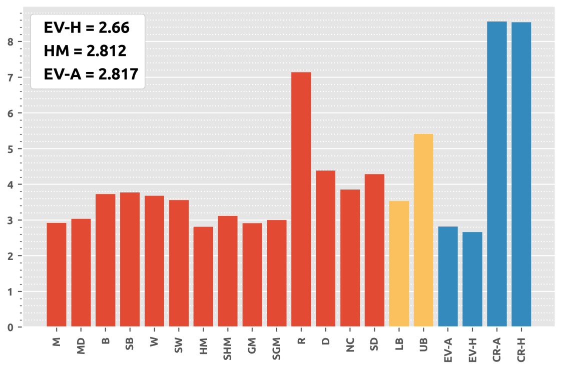
(a) $V = 0.2$, $\mu_{\text{ccr}} = 0.01$.



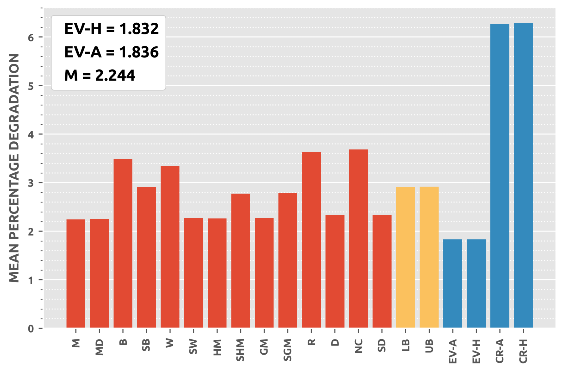
(b) $V = 1.0$, $\mu_{\text{ccr}} = 0.01$.



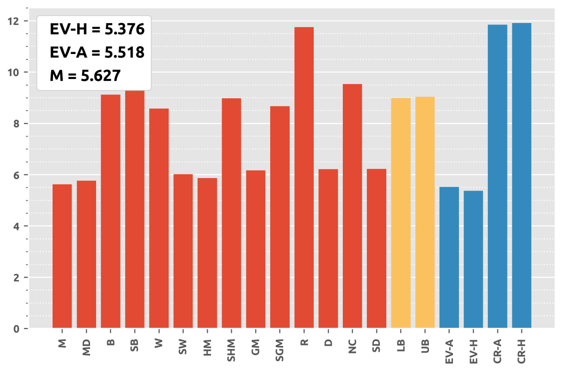
(c) $V = 0.2$, $\mu_{\text{ccr}} = 0.1$.



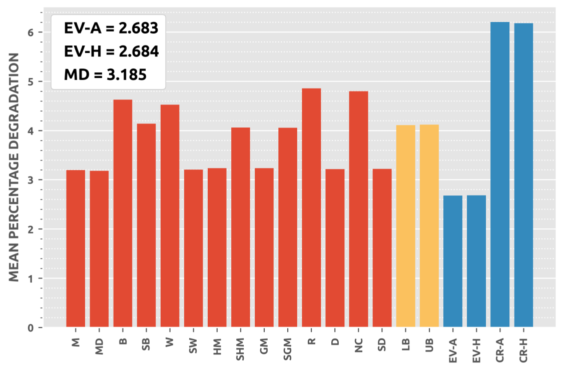
(d) $V = 1.0$, $\mu_{\text{ccr}} = 0.1$.



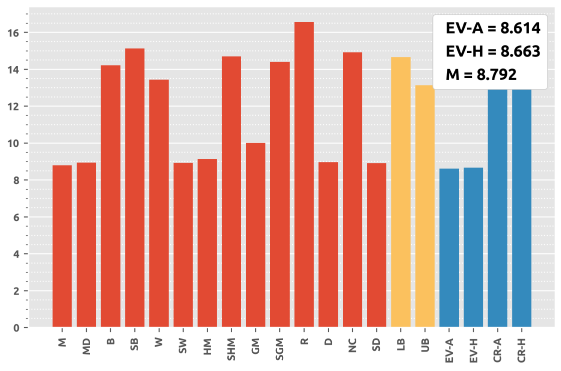
(e) $V = 0.2$, $\mu_{\text{ccr}} = 1.0$.



(f) $V = 1.0$, $\mu_{\text{ccr}} = 1.0$.



(g) $V = 0.2$, $\mu_{\text{ccr}} = 2.0$.



(h) $V = 1.0$, $\mu_{\text{ccr}} = 2.0$.

Figure 3.4: Mean percentage degradation (MPD) of task prioritization schemes for STG set with different combinations of V (coefficient of variation) and μ_{ccr} (mean CCR). Legends identify the three best. Red bars represent averaging-based schemes, yellow those based on bounds on the critical path length, and blue those derived from the stochastic interpretation described in Section 3.3.3.

slightly cheaper than EV-H) was at least 5 times as expensive as the averaging methods. As discussed in Section 3.3.3, this may be better than expected, given that we used 1000 realizations in the MC method, but it could still be prohibitively high depending on the context; this is a determination that will likely always have to be made by users based on their own needs.

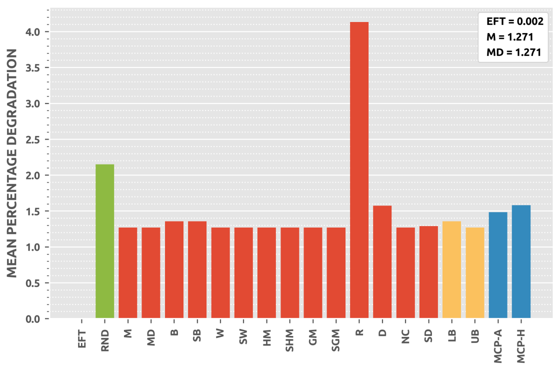
(We have not so far addressed one obvious question: do we actually need 1000 MC realizations in order to get a good approximation of the expected value? Could, say, 10 realizations be nearly as accurate and significantly cheaper? In fact, we found that the performance of EV-A/H was almost as good with 10 realizations as with 1000. However, because our code is largely vectorized, the former was not significantly more expensive than the latter for the relatively small graphs from the STG, although the disparity was more pronounced for the larger Cholesky factorization graphs. Therefore it seemed sensible to use 1000 realizations as that gives a tighter approximation. The broader question of how many MC realizations are typically required in practice to approximate the critical path distribution will be treated in greater depth in the next chapter.)

3.4.3 Critical path assignment

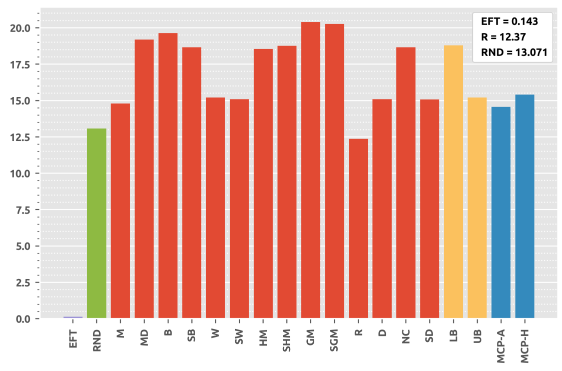
In an attempt to determine how the critical path should be approximated in the *assignment* framework, we compared the 18 different methods for identifying the critical path of a graph that were described in Section 3.3: the 14 averaging schemes and the 2 bounds, plus the most frequently critical path that we observed from the MC method for the two different sets of pmfs (A or H, as in the previous section). The averages and bounds are referred to as in the previous section. The two MC-based methods are denoted by MCP-A and MCP-H (for *most critical path*). In addition, we included two other methods in the comparison, to make a total of 20. The first is referred to as EFT since it is defined by using the earliest finish time (EFT) rule to schedule *all* tasks—i.e., not using critical path assignment at all. This is equivalent to the standard HEFT algorithm since, as noted earlier, we always used the standard upward ranks to prioritize the tasks. EFT was included to answer the question, when is scheduling the expected critical path on a single processor actually helpful at all? The other additional method is denoted by RND and was likewise included to gauge the effectiveness of path assignment as a concept. It is defined by assigning a randomly chosen path (which was computed by simply beginning at the source and repeatedly selecting a child task at random until the sink was reached).

Figure 3.5 illustrates the MPD of each method for the Cholesky graphs. We see that the picture is very much mixed: EFT was utterly dominant for $\beta = 0.01$ and 0.1 but among the worst for three of the four other subsets. The explanation for this is straightforward. Even assuming that we can successfully identify the critical path, it is only sensible to schedule its tasks on a single processor when the time saved made by eliminating communication between them is greater than the time saved by scheduling the tasks on the processors which minimize their execution times (consider the simplest example of a graph comprising a single chain of tasks). Therefore we can only expect critical path assignment to be helpful when communication is not dwarfed by computation. Moreover, it seems foolish to compare the different methods to one another when they are all objectively poor. (Indeed, if we follow the chain of logic further it could even be argued that a method which does worse than another with low communication costs may actually be *more successful* in identifying the critical path since poor scheduling of its path leads to worse performance than poor scheduling of the other's.) However, even if we restrict ourselves only to those parameter combinations for which EFT does not dominate, it is difficult to see any clear trends in Figure 3.5. For example, when $\beta = 10$, RND is one of the best with $V = 1$ but one of the worst for $V = 0.2$. In fact, RND was the best on average for the entire set of Cholesky graphs.

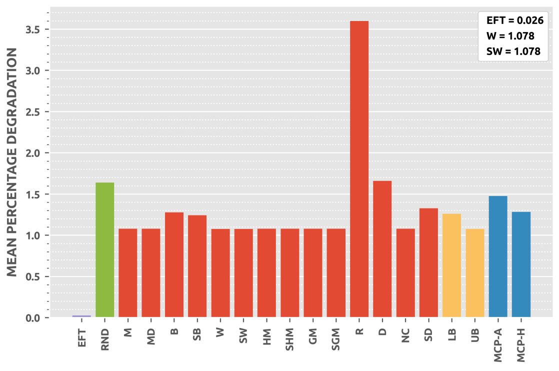
Critical path assignment was even less effective for the graphs from the STG set. We found that, for every subset of the graphs defined by a combination of parameters in the cost-setting algorithm, EFT was always the best on average. This was true even for $\mu_{\text{ccr}} = 2$, when communication was highest. However, interesting behavior is apparent if we consider the subset of 62140 graphs ($\approx 53\%$ of the total) for which at least one of the path assignment methods recorded a shorter schedule than EFT. Figure 3.6 shows the percentage of the 62140 graphs for which each of the methods was the best. We see that there is no dominant method: each was the best at least 15% of the time and the highest percentage recorded was just over 20% (by R). The interesting part is that the preference-based averaging methods R, D, SD and NC were the best, when, with the exception of SD, they have typically been among the worst for task prioritization, both in this chapter and the previous. It isn't clear whether this is truly meaningful or if it is simply an artifact of



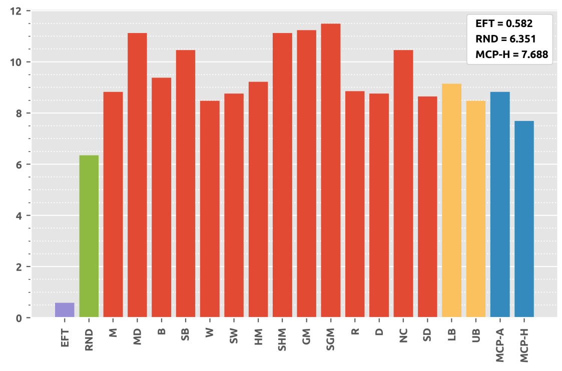
(a) $V = 0.2$, $\beta = 0.01$.



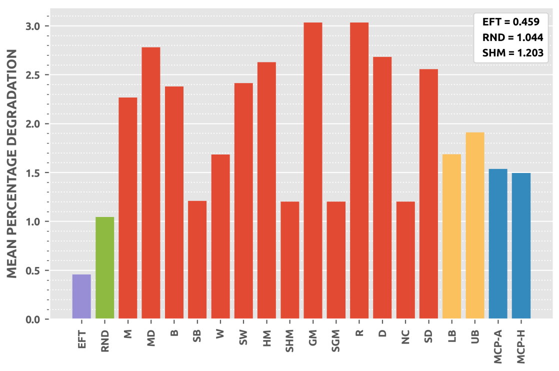
(b) $V = 1.0$, $\beta = 0.01$.



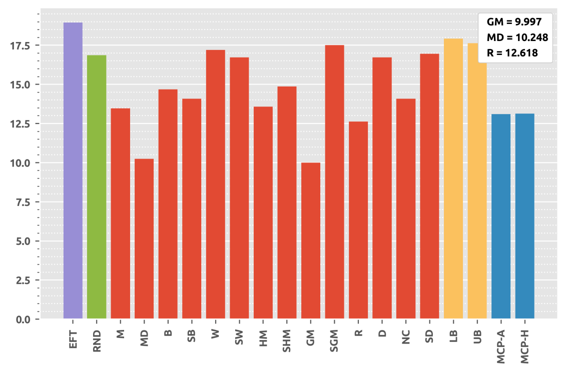
(c) $V = 0.2$, $\beta = 0.1$.



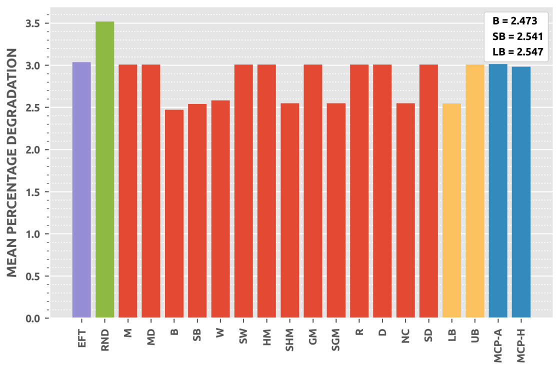
(d) $V = 1.0$, $\beta = 0.1$.



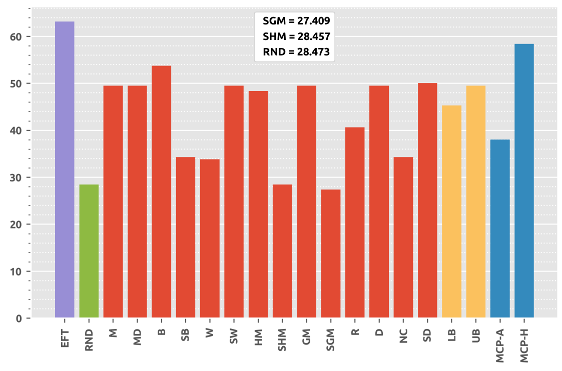
(e) $V = 0.2$, $\beta = 1.0$.



(f) $V = 1.0$, $\beta = 1.0$.



(g) $V = 0.2$, $\beta = 10.0$.



(h) $V = 1.0$, $\beta = 10.0$.

Figure 3.5: Mean percentage degradation (MPD) of path assignment schemes for Cholesky graphs with different combinations of V (coefficient of variation) and β (the CCR). Legends identify the three best. Red bars represent averaging-based schemes, yellow those based on bounds on the critical path length, and blue those derived from the stochastic interpretation described in Section 3.3.3. The purple bar represents no path assignment and the green assigning a randomly-chosen path.

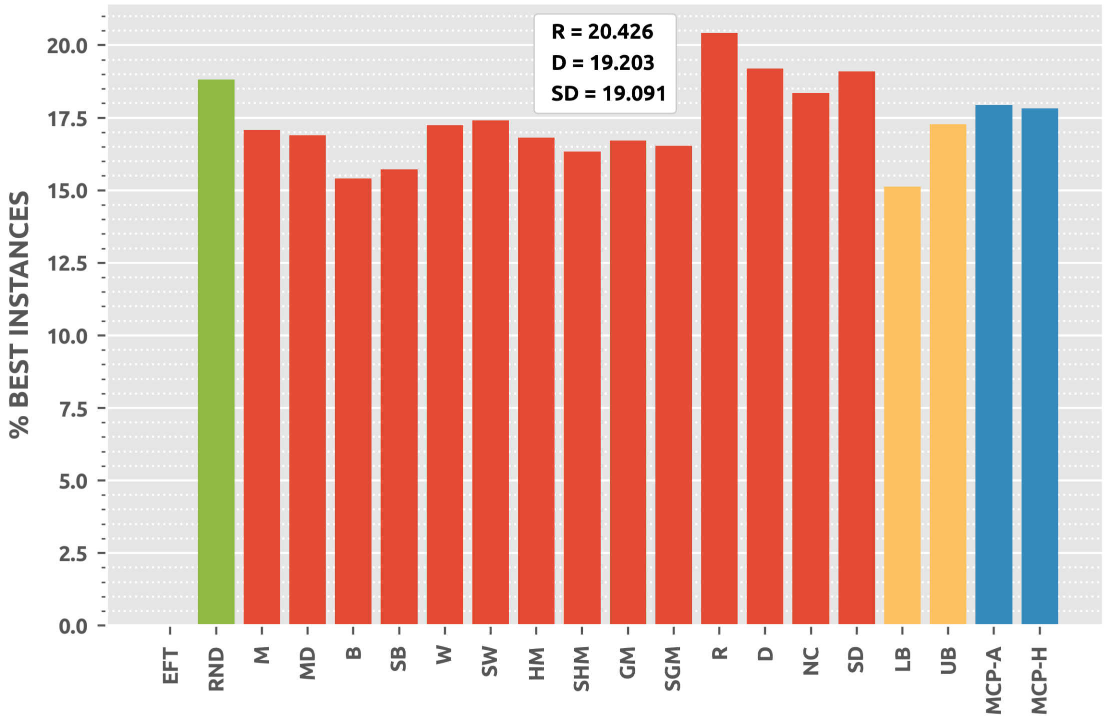


Figure 3.6: Percentage of the 62140 graphs from the STG for which EFT was *not* the best that each of the path assignment methods were instead (so that higher values indicate better performance). Legend identifies the three best. Ties are not distinguished so sum of percentages may exceed 100. Red bars represent averaging-based schemes, yellow those based on bounds on the critical path length, and blue those derived from the stochastic interpretation described in Section 3.3.3. The green bar represents assigning a randomly-chosen path.

the fact that the graphs considered are those for which EFT did relatively poorly. Indeed, it is very difficult to draw any firm conclusions here; it seems safest to say that identifying the best critical path for assignment is a subtler problem than it may first appear.

3.5 Conclusions and future work

As in the previous chapter, our foremost takeaway from the empirical investigation conducted here was a greater appreciation for the difficulty of the DAG scheduling problem in heterogeneous computing. We repeatedly found that evaluating results was difficult, with many possible interpretations. Regarding the task prioritization usage of the critical path in particular, we make the following conclusions.

1. The new methods EV-A and EV-H that were proposed in Section 3.3.3 performed very well for the graphs from the STG set, achieving better schedules than all the

alternatives on average. They were more expensive, however, so schedule makespan reductions may be outweighed by the additional time required to compute them. Furthermore, although still broadly competitive, various averaging-based methods were superior for the Cholesky graphs when the cost heterogeneity was high (i.e., $V = 1$).

2. The prioritization methods CR-A and CR-H based on computing task criticalities using MC were consistently the worst. It is not clear why this is the case, although the fundamental issues with using criticality to prioritize tasks that were highlighted by Williams [139] may be responsible.

3. The LB and UB rankings, based on the critical path bounds described in Section 3.3.2, were mediocre and in particular did not improve on the B/SB and W/SW averaging schemes that they were intended to extend.

4. Results were again granular, with the best method for a task graph and target platform appearing to depend on a wide range of factors. This suggests that the wisest course of action in practice is to use different methods in different situations, guided by exploratory testing similar to the investigation that was described here.

Our conclusions concerning the path assignment usage of the critical path reflect the previous point. Determining the best method is clouded by the fact that the concept itself only appears to be useful when communication predominates computation. However, even when this is the case, it is far from clear which methods are the best and when, with none of those considered here distinguishing themselves above others. In terms of future research, the most obvious extension of this work would be to increase the scale of the empirical investigation, including considering more graphs from real-world applications.

CHAPTER 4

PREDICTING SCHEDULE LENGTH UNDER UNCERTAINTY

Suppose that we have computed a schedule π for a task graph G. What is the makespan of π? Assuming that all task execution times and communication delays—the *schedule costs*—are known exactly, this is straightforward. But what if the actual schedule costs cannot be predicted precisely? What if we know that there are a range of possible values each may take at runtime? In practice, this will almost always be the case: typically, the best that we can do is to define some probability distributions that we believe they follow, based either on theoretical analysis or relevant data (such as the BLAS kernel benchmarking described in Section 2.5.1). In other words, the costs can be modeled as *random variables* (RVs), rather than fixed scalars. But if the schedule costs are stochastic then clearly the makespan itself must be as well; how do we compute its distribution?

4.1 A LONGEST PATH PROBLEM

Of course, if schedule costs are RVs rather than scalars, it is impossible to specify the precise times at which processors should execute their assigned tasks. Therefore, at this juncture we should define precisely what we mean by a *schedule*: we assume that a schedule π is a mapping from tasks to processors that specifies only which tasks each processor should execute and the order in which this should be done, with the understanding that the processor executes its next scheduled task as soon as it is able—i.e., without artificial delays. Conceptually, we can view this as all processors being assigned an ordered queue

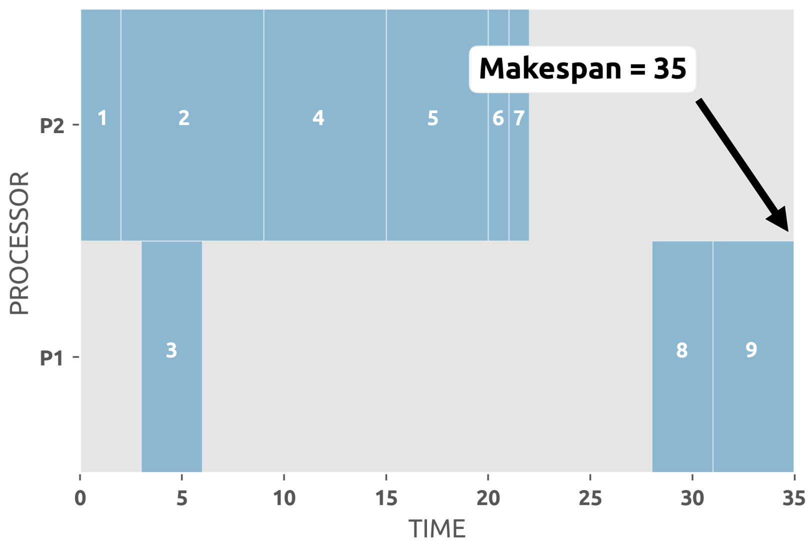


Figure 4.1: A schedule for the task graph from Figure 3.1. Displayed in this manner, the makespan is clear. Note that although there is a gap in which both processors are idle due to communication delays, this is in fact an optimal schedule.

of tasks before runtime and only being allowed to execute the task currently at the head of their queue.

Since it defines the execution order of tasks on processors, any schedule π for an application with task graph G can therefore be represented by another graph G_π which contains all the same nodes and edges as G, plus an additional set of *disjunctive* edges that indicate the execution order of the tasks on their chosen processors. We will refer to G_π as the *schedule graph* associated with the task graph G and schedule π. Note that there is some flexibility in how we add the disjunctive edges but the most straightforward way is to simply add a disjunctive edge between a given task and the task which is executed immediately before it on the same processor if no edge already exists between the two. To illustrate the process, Figure 4.2 presents a schedule graph for the schedule from Figure 4.1 and task graph from Figure 3.1. Observe that we only need to add three disjunctive edges to the task graph topology in order to construct the schedule graph.

The weights of G_π are induced by the processor selections of π, assuming that all disjunctive edges have weight zero. In particular, this means that the longest path of G_π is equal to $|\pi|$, the makespan of π. For example, we see in Figure 4.2 that the longest path is $(1, 2, 4, 5, 6, 7, 9)$, with a length of 35, which is the schedule makespan. Therefore, finding the distribution of the makespan for a schedule with stochastic costs is equivalent to computing the distribution of the longest path through a DAG with stochastic weights.

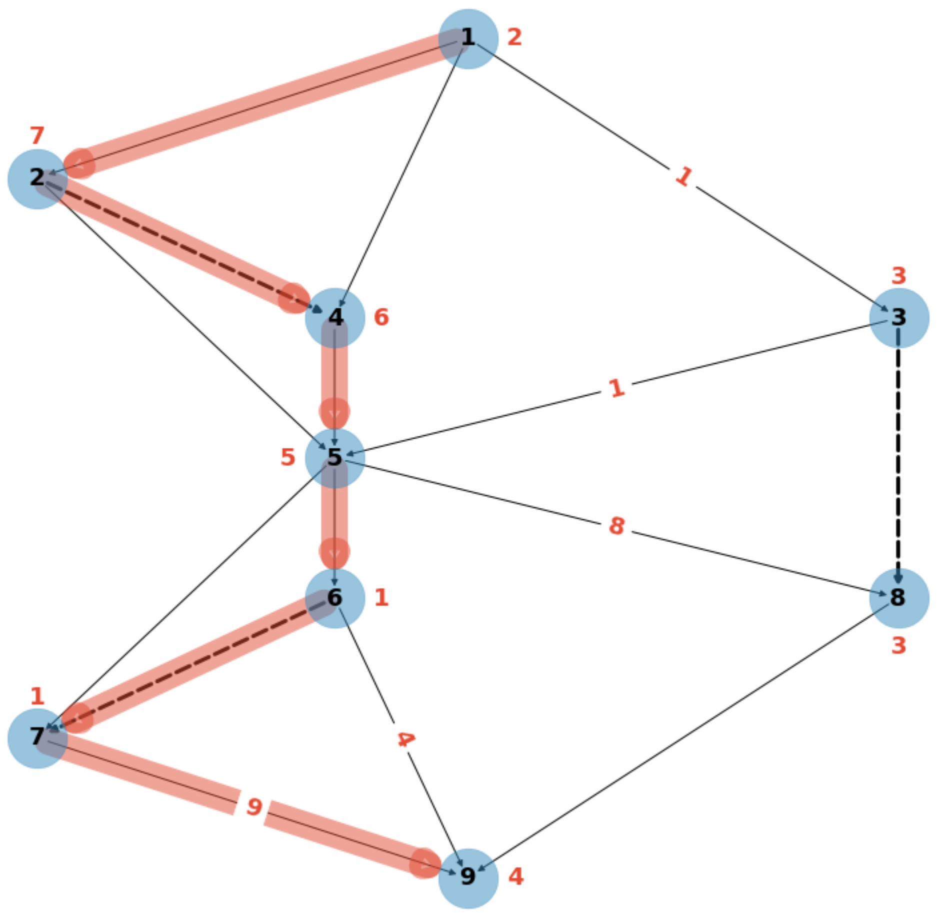


Figure 4.2: Schedule graph corresponding to schedule from Figure 4.1 and task graph from Figure 3.1. Task execution costs under the schedule are denoted by the red labels near the nodes. Unlabeled edges have zero cost, including disjunctive edges which are indicated by the dotted lines. Red highlighted edges comprise the longest path.

4.1.1 Complexity

For a generic task t_i, let w_i be its execution cost according to the schedule π—and therefore the weight of t_i in the schedule graph G_π. Similarly, let w_{ik} be the weight of a generic edge (t_i, t_k) in G_π. If the schedule costs were scalars, as in the example above, then finding the longest path through G_π would be straightforward. In particular, we would recursively compute a sequence of numbers L_i defined by $L_1 = w_1$ and

$$L_i = w_i + \max_{h \in \Gamma_i^-} \{w_{hi} + L_h\} \tag{4.1}$$

for all other $i = 2, \dots, n$. The longest path of G_π, and therefore the makespan of π, would then be given by L_n. Computing these values is an $O(n+e) \approx O(n^2)$ operation, which depending on the size of the DAG may be expensive but is at least polynomial. (Of course, we could work backward through the DAG by setting $L_n = w_n$ and doing the maximization over the set of task children in Eq. (4.1) instead; the makespan would then be given by L_1 but the procedure is otherwise equivalent.)

However, if the schedule costs w_i and w_{ik} are RVs instead of scalars, then it isn't clear how we can apply Eq. (4.1) in order to compute the distribution of the schedule makespan. Fundamentally, Eq. (4.1) comprises two operations: summation and maximization. Neither of these are straightforward for arbitrary RVs. If we assume that the cost RVs are independent of one another, then summations and maximizations can be computed through convolutions and products of their distribution functions, respectively (see Section 4.2.2). But this presupposes that the distributions of all weight RVs are fully known, which may not be the case. Moreover, observe that the maximization is effectively computed for a set of RVs which represent path lengths—i.e., summations of task and edge weight RVs—and, even if all of the individual weights are independent of one another, the lengths of two or more paths typically won't be independent because of shared tasks and edges. This means that maximizations will typically need to be performed for sets of dependent RVs at any rate. Formalizing the intuitive difficulty, Hagstrom proved that computing the longest path distribution of a graph with stochastic weights, or even just its expected value, is a $\#P$-complete[1] problem for discrete RVs [60].

[1] The traditional way to explain the complexity of this class intuitively is to say that solving a $\#P$-complete problem is equivalent to counting the number of solutions to an NP-complete one [132].

4.1.2 Scope of this chapter

The problem of computing the distribution of the longest path through a DAG with stochastic weights was first studied in the context of *Program Evaluation Review Technique* (PERT) [79] network analysis. A PERT network is essentially what we have referred to here as a schedule graph, with the graph representing a project and its weights the time that its constituent tasks will take to complete. But the stochastic longest path problem has also been studied in other research areas such as digital circuit design [24]. Therefore, from now on, we will explicitly focus on the more general problem of finding the longest path distribution for a graph with stochastic weights, rather than the makespan of a schedule with stochastic costs. However, as described above, the two are equivalent.

We make only one significant assumption about the graph weights: namely, that they are *independent*. Realistically, this is unlikely to be wholly true for schedule graphs. For example, uncertainty in task execution times may be due at least in part to fluctuations in processor speeds; if task t_i takes longer than expected on processor p_a, it is likely that the next task t_k scheduled on p_a will also take longer than average. But the independence assumption is not completely unrealistic either. Moreover, it makes the problem slightly more tractable and is common in the literature [34], [112]. Beyond independence, the only other assumption we make about the weight distributions is that we can at least estimate their means and standard deviations. This is broadly reasonable; for example, assuming we have access to benchmarking data such as that gathered in Section 2.5.1, we can use sample summary statistics as estimators.

Given the difficulty, computing the longest path distribution exactly is typically impractical, so approximations are needed instead. In the remainder of this chapter, we give a brief overview of various bounds and heuristics that have been employed for this purpose, before proposing a heuristic framework of our own for tackling the problem.

4.2 BOUNDS

For all $i = 1, 2, \dots n$, let L_i be the RV representing the length of the longest path from the source t_1 to the task t_i, so that in particular L_n represents the distribution of the longest path through the entire DAG. Although computing L_n exactly is usually impractical, bounds on various quantities of interest may be computed much more cheaply. Depending on the context, these may be tight enough to be useful, a claim that we will investigate empirically for certain examples in Section 4.5.

4.2.1 On the moments

Some of the oldest results concern the expected value of the longest path. Indeed, as we have already seen in the previous chapter, a lower bound which dates back to the earliest days of PERT analysis can be computed in $O(n^2)$ operations by replacing all weight RVs with their expected value and then computing the longest path through the resulting scalar-weight graph in the same manner as Eq. (4.1). Define $u_1 = \mathbb{E}[w_1]$ and recursively compute

$$u_i = \mathbb{E}[w_i] + \max_{h \in \Gamma_i^-}\{\mathbb{E}[w_{hi}] + u_h\} \tag{4.2}$$

for all other $i = 2, \dots, n$. Then we have $u_i \leq \mathbb{E}[L_i]$ for all i, so that in particular $u_n \leq \mathbb{E}[L_n]$. We will refer to this as the *Critical Path Method* (CPM) bound [79].

Fulkerson's bounds and extensions. An alternative number sequence f_i which yields tighter bounds on the expected value of the longest path than the u_i numbers was given by Fulkerson [55]. Suppose, for the moment, that all of the weights follow discrete distributions. For all $i = 1, \dots, n$, define Z_i to be the set of all weight RVs corresponding to nodes and edges between the source and task t_i. Let $R(Z_i)$ be the set of all possible realizations of the RVs in Z_i. Given a realization $z_i \in R(Z_i)$, let $\ell(z_i)$ be the length of the longest path from the source to task t_i (which is a scalar because all weights have been realized). Then by the definition of the expected value we have

$$\mathbb{E}[L_i] = \sum_{z_i \in R(Z_i)} \mathbb{P}[Z_i = z_i]\ell(z_i). \tag{4.3}$$

Let B_i be the set of all the weight RVs corresponding to task t_i's immediate parents and the edges connecting them to t_i. Further, let $R(B_i)$ be the set of all possible realizations of the RVs in B_i and let $b_i \in R(B_i)$ be any such realization. Note that we can break up Eq. (4.3) by recursively considering the immediate parents of a given task and rewrite it as

$$\mathbb{E}[L_i] = \sum_{b_i} \sum_{\substack{z_h \in R(Z_h) \\ h \in \Gamma_i^-}} \mathbb{P}[B_i = b_i]\mathbb{P}[Z_h = z_h] \max_{h \in \Gamma_i^-}\{\ell(z_h) + b_{hi}\},$$

where b_{hi} is the realization of the edge weight RV w_{hi} according to the set of realizations b_i. Now suppose we recursively define a sequence of numbers by $f_1 = \mathbb{E}[w_1]$ and

$$f_i = \sum_{b_i \in R(B_i)} \mathbb{P}[B_i = b_i] \max_{h \in \Gamma_i^-} \{f_h + b_{hi}\}, \tag{4.4}$$

otherwise. Then Fulkerson proved that $u_i \leq f_i \leq \mathbb{E}[L_i]$ holds for all $i = 1, \dots, n$.

Fulkerson's bounds were extended to continuous weights by Clingen [39], who also suggested a more computationally tractable formulation of Eq. (4.4), assuming that all the graph weights are independent (as we have done here by default). The bounds were later tightened by Elmaghraby [52], with the usual downside of greater computational effort being required, before the entire approach was generalized by Robillard and Trahan [102].

Upper bounds on the mean. The methods discussed above provide only lower bounds for the expected value. However, Dodin's [47] lower bound on the distribution function also induces an upper bound on the expected value (see below). Furthermore, if all graphs weights follow *New Better Than Used in Expectation* (NBUE)[2] distributions, then upper bounds can be computed by substituting all weights with exponentially distributed RVs that have the same expected values [65], [143]. This approach has the advantage that only the expected values of all weights are necessary, rather than their full distributions.

Normal weights. If the task and edge weights are all normally distributed, then Kamburowski [66] was able to prove both lower and upper bounds on the expected value. Furthermore, he also proved a lower bound on the variance and conjectured an upper bound. These are the only non-trivial bounds on moments higher than the first that we are aware of, although the upper is unproven and, as we shall see later, the lower is typically very loose.

The basic idea is to recursively compute four number sequences $\underline{m_i}$, $\overline{m_i}$, $\underline{s_i}$ and $\overline{s_i}$ such that $\underline{m_i} \leq \mathbb{E}[L_i] \leq \overline{m_i}$ and $\underline{s_i}^2 \leq \mathrm{Var}[L_i] \leq \overline{s_i}^2$ for all $i = 1, \dots, n$. Clearly, by taking

$$\underline{m_1} = \overline{m_1} = \mathbb{E}[w_1]$$

and

$$\underline{s_1}^2 = \overline{s_1}^2 = \mathrm{Var}[w_1]$$

[2]A concept from reliability theory, referring to distributions representing object lifetimes such that, at any given time, the expected value of the remaining lifetime is smaller than the expected value of the entire lifetime. Certain common distributions such as Erlang and uniform are NBUE, as are many others such as gamma under restricted parameter regimes.

we can achieve the desired bounds for L_1. (As ever, we could work backward instead, in which case the analogous results hold for the index n.) Now suppose that we move forward through the DAG in a topologically sorted order. For a generic task t_i, the variance bounds are relatively straightforward, albeit loose. The lower bound is given by

$$\underline{s_i^2} = \begin{cases} \underline{s_h^2} + \mathrm{Var}[w_{hi}] + \mathrm{Var}[w_i], & \text{if } \Gamma_i^- = \{t_h\}, \\ 0, & \text{otherwise,} \end{cases} \tag{4.5}$$

reflecting the intuition that the variance can be reduced almost arbitrarily by a maximization (which needs to be performed in the case of multiple parents). The upper bound is defined through

$$\overline{s_i^2} = \max_{h \in \Gamma_i^-} \left\{ \overline{s_h^2} + \mathrm{Var}[w_{hi}] + \mathrm{Var}[w_i] \right\}. \tag{4.6}$$

When the graph weights are all independent and normally distributed, the variance of a path length is just the sum of the variances of the weights RVs along it. Therefore the right-hand side of Eq. (4.6) is the maximum variance of any path from the source to t_i. Intuitively, then, the bound is equivalent to the proposition that the variance of the maximum of a set of (dependent) normal RVs is bounded above by the greatest variance of the maximands; see Section 4.4.2 for more on this.

The bounds on the expected value are somewhat less intuitive. Let

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \quad \text{and} \quad \Phi(x) = \int_{-\infty}^{x} \phi(t)\,\mathrm{d}t \tag{4.7}$$

be the unit normal probability density function and cumulative probability function, respectively. Define a function h by

$$h(\mu_i, \sigma_i, \mu_k, \sigma_k) = \mu_i \Phi(\delta) + \mu_k \Phi(-\delta) + \gamma \phi(\delta),$$

where $\gamma = \sqrt{\sigma_i^2 + \sigma_k^2}$ and $\delta = (\mu_i - \mu_k)/\gamma$. Suppose that we have a set of (not necessarily independent) normally distributed RVs $X_1, X_2, \ldots, X_r$, where each $X_i \sim \mathrm{N}(\mu_i, \sigma_i^2)$ and $\sigma_1 \leq \sigma_2 \leq \cdots \leq \sigma_r$. Define two functions $\underline{f}$ and $\overline{f}$ by the recursions,

$$\begin{aligned}
\underline{f}(X_1) &= \overline{f}(X_1) = \mu_1, \\
\underline{f}(X_1, X_2) &= \overline{f}(X_1, X_2) = h(\mu_1, \sigma_1, \mu_2, \sigma_2), \\
\underline{f}(X_1, \ldots, X_r) &= h\big(\underline{f}(X_1, \ldots, X_{r-1}), 0, \mu_r, \sigma_r\big), \\
\overline{f}(X_1, \ldots, X_r) &= h\big(\overline{f}(X_1, \ldots, X_{r-1}), \sigma_{r-1}, \mu_r, \sigma_r\big).
\end{aligned}$$

Then Kamburowski proved that, if we define

$$\underline{m_i} = \underline{f}\left(\{\underline{X_h}\}_{h\in\Gamma_i^-}\right) \quad \text{and} \quad \overline{m_i} = \overline{f}\left(\{\overline{X_h}\}_{h\in\Gamma_i^-}\right)$$

for all $i = 2, \dots, n$, where

$$\underline{X_h} \sim \mathrm{N}\left(\underline{m_h} + \mathbb{E}[w_{hi}] + \mathbb{E}[w_i],\ \underline{s_h^2} + \mathrm{Var}[w_{hi}] + \mathrm{Var}[w_i]\right),$$
$$\overline{X_h} \sim \mathrm{N}\left(\overline{m_h} + \mathbb{E}[w_{hi}] + \mathbb{E}[w_i],\ \overline{s_h^2} + \mathrm{Var}[w_{hi}] + \mathrm{Var}[w_i]\right),$$

and the sets are ordered in such a way that the inequality constraints on the variances is satisfied, we have

$$\underline{m_i} \leq \mathbb{E}[L_i] \leq \overline{m_i}.$$

Furthermore, he also showed that $u_i \leq \underline{m_i}$ holds for all i—i.e., the lower bound is tighter than the traditional CPM bound.

Although these bounds strictly only hold for normally distributed weights, it is possible they may be useful approximations in other cases, particularly since they are conceptually similar to the heuristics for computing the longest path distribution that are described in Section 4.3.2. Therefore we investigate this possibility in Section 4.5.

4.2.2 On the distribution function

Rather than just the moments, we may be more interested in bounds on F_n, the cumulative distribution function of the longest path. In this context, a bound indicates (first-order) *stochastic dominance* between the distributions; in particular, B stochastically dominates L_n if

$$F_n(z) = \mathbb{P}[L_n \leq z] \leq \mathbb{P}[B \leq z] = F_B(z), \quad \forall z.$$

For convenience, from now on we simply write $F_n \leq F_B$ to represent this expression, so that the aim here is to determine F_b and F_B such that $F_b \leq F_n \leq F_B$.

The first bounds on the distribution of the longest path were provided by Kleindorfer [69]. As stated earlier, distribution functions for the sum and maximization of *independent* random variables are easily computed. In particular, suppose X and Y are independent, then F_{X+Y} is computed through the convolution

$$F_{X+Y}(z) = \int_{-\infty}^{\infty} F_Y(z-t) f_X(t)\ \mathrm{d}t, \tag{4.8}$$

where f_X is the probability density function of X, and $F_{\max(X,Y)}$ via the product

$$F_{\max(X,Y)}(z) = F_X(z) \times F_Y(z). \tag{4.9}$$

Both of these can be readily approximated through standard numerical methods. Kleindorfer's upper bound is given by working through the graph in the usual manner and assuming path independence—i.e., computing Eq. (4.1) using Eqns. (4.8) and (4.9) for sums and maximizations, respectively. A corresponding lower bound is given by effectively disregarding all maximizations and simply choosing one of the maximands at random.

Inspired by the observation that Kleindorfer's upper bound is exact when all path lengths are independent, Dodin [47] combined the method with a sequence of transformations such that the graph is reduced to a single edge whose associated distribution function bounds the longest path distribution from above. If the graph is *series-parallel* then the bound is exact. Moreover, Dodin showed that his bound is tighter than Kleindorfer's, and, as noted above, that a corresponding lower bound on the expected value can be inferred.

Empirical studies by Ludwig, Möhring and Stork [78] and Canon and Jeannot [34] suggest that the bounds of Kleindorfer and especially Dodin are usually tight, so they may be useful as approximations of the distribution function in addition to formal bounds. However, although both methods run in polynomial time, they tend to be more expensive than heuristic solutions without pronounced improvements in accuracy, so are of less interest from a practical scheduling perspective.

4.3 HEURISTICS

Although bounds obviously may be useful, in practice we are often satisfied with good approximations to the longest path distribution. In a scheduling context, for example, it may be the case that there are many prospective schedules that we need to quickly evaluate in order to select the best among them. Many heuristics for estimating the longest path distribution have therefore been proposed. In this section we briefly outline some noteworthy examples.

4.3.1 Monte Carlo simulation

The Monte Carlo (MC) method has a long history in approximating the longest path distribution of PERT networks, dating back to at least the early 1960s [133]. As introduced in the previous chapter, the basic idea is to repeatedly simulate the realization of all weight RVs and then evaluate the longest paths of the resulting graphs. This procedure gives us a set of longest path instances whose empirical distribution function is guaranteed to converge to the true distribution by the Glivenko-Cantelli Theorem[3]. This represents a considerable advantage compared to the family of heuristics introduced in the next section, which only estimate the first two moments of the distribution. Furthermore, we can (at least roughly) quantify the approximation error for any given number of samples. For example, confidence intervals for the sample mean and variance can easily be constructed [133]. On top of these theoretical assurances, the other big advantage of the MC method is its simplicity: once the graph weights are realized, we only need to perform sums and maximizations of scalar values, rather than RVs.

There are however also downsides to the MC method. First, all weight distributions must be known in order to accurately simulate realizations. From a scheduling perspective, this is not always the case for the execution times that task weights represent, let alone the communication delays represented by the edge weights. On the other hand, it could be argued that, since the Central Limit Theorem says that path lengths tend to normality as they grow, so long as weight means and variances are known, the exact distributions they follow become less important once the graph becomes sufficiently deep. Therefore, assuming the graph is large enough, sampling weight realizations from any distributions with the prescribed means and variances will likely result in a good approximation of the actual longest path distribution; this hypothesis will be evaluated via simulation later.

The second, more critical, drawback of MC simulation is the computational cost. For each complete realization of the DAG weights, we need to compute the longest path, which is an $O(n^2)$ operation. The total cost is therefore $O(Rn^2)$, where R is the number of graph realizations. Clearly, if the desired value of R is large then this may be impractical. Again, however, there are caveats to this warning: as we saw in the previous chapter, in practice it can often be the case that only a relatively small number of realizations are

[3]Despite this convergence in the limit, we refer to MC as a heuristic since any solutions obtained in practice will only ever be approximate.

actually required to obtain good estimates of the longest path distribution, even for very large graphs. Furthermore, the MC method can easily be parallelized, suggesting that runtimes may be more reasonable in reality than the analysis might indicate. Various investigations along these lines are detailed in Section 4.5.

Given that the MC method is traditionally viewed as expensive, reducing the computational effort without sacrificing too much accuracy has long been desirable. Indeed, Van Slyke's [133] original paper introducing the MC method for PERT networks devotes time to this, suggesting that it may be beneficial to initially perform a small number of realizations of the graph in order to estimate which tasks have very low *criticality*—the probability of being on a longest path, as defined in the previous chapter—before removing them from the graph and performing more iterations of the MC method on the reduced graph. Van Slyke also outlined an analytical approach to identifying tasks which are highly unlikely to lie on critical paths, assuming that all weight distributions have finite ranges. However, the method can fail to identify all tasks which can never be critical and may be expensive besides. More sophisticated analytical techniques that aim to reduce the overall cost by minimizing the number of RVs which need to be sampled were suggested by Burt and Garman [31], Sigal, Pritsker and Solberg [116], and later Fishman [54]. The drawback of all of these is that the initial analysis may be more expensive than simply doing the traditional MC method, especially for complex graph topologies.

4.3.2 The Central Limit Theorem

Fundamentally, the length of any given path through a weighted graph is just the sum of the individual weights along it. By the Central Limit Theorem (CLT), sums of independent random variables are asymptotically normally distributed. So we can make a reasonable argument that the longest path distribution itself may be at least approximately normal, $L_n \approx \mathrm{N}(\mu_n, \sigma_n)$. This intuition forms the basis of a family of efficient heuristics for computing an approximation to the longest path distribution.

Clark's equations. If we assume that all weight RVs can be characterized by their mean and variance (i.e., effectively that they are also normal), then sums can be computed though the well-known rule for any two normal RVs $\epsilon \sim \mathrm{N}(\mu_\epsilon, \sigma_\epsilon^2)$ and $\eta \sim \mathrm{N}(\mu_\eta, \sigma_\eta^2)$,

$$\epsilon + \eta \sim \mathrm{N}\left(\mu_\epsilon + \mu_\eta, \sigma_\epsilon^2 + \sigma_\eta^2 + 2\rho_{\epsilon\eta}\sigma_\epsilon\sigma_\eta\right), \tag{4.10}$$

where $\rho_{\epsilon\eta}$ is the linear correlation coefficient between the two distributions (assumed to be zero here since they are independent). Formulae for the first two moments of the maximization of two normal RVs—which is not itself normal—are less well-known but were first provided by Clark in the early 1960s [38]. Let ϕ and Φ be the unit normal probability density and cumulative probability functions, as defined by Eq. (4.7). Furthermore, for ϵ and η as above, define

$$\alpha = \sqrt{\sigma_\epsilon^2 + \sigma_\eta^2 - 2\rho_{\epsilon\eta}\sigma_\epsilon\sigma_\eta} \quad \text{and} \quad \beta = \frac{\mu_\epsilon - \mu_\eta}{\alpha}. \tag{4.11}$$

Then the first two moments $\mu_{\max}$ and $\sigma_{\max}^2$ of $\max(\epsilon,\eta)$ are given by

$$\mu_{\max} = \mu_\epsilon\Phi(\beta) + \mu_\eta\Phi(-\beta) + \alpha\phi(\beta), \tag{4.12}$$

$$\sigma_{\max}^2 = (\mu_\epsilon^2 + \sigma_\epsilon^2)\Phi(\beta) + (\mu_\eta^2 + \sigma_\eta^2)\Phi(-\beta) + (\mu_\epsilon + \mu_\eta)\alpha\phi(\beta) - \mu_{\max}^2. \tag{4.13}$$

Although these formulae are exact, they are only valid for a single pair of normal RVs. The maximization of two normal RVs is not itself normal, so we cannot obtain the exact mean and variance for the maximum of arbitrarily many RVs by applying them in a pairwise manner. However, we can at least get an approximation. Sinha, Zhou and Shenoy [120] empirically investigated the accuracy of this approximation for sets of up to 100 normal RVs, ultimately concluding that it is usually good, with the average approximation error in the mean and standard deviation around 2% and 14%, respectively. Furthermore, they considered several possible orderings for the operands of the maximization (randomly, sorted by mean value, and so on), a topic that was also briefly discussed by Ross [104]. They found that significant improvements in accuracy could sometimes be made through different orderings. However, the best-performing choices that they considered were also the most expensive, so in our own implementation, used in the simulations described later, we decided to simply treat the maximands in a random order.

Sculli's method. By using Eq. (4.10) for summations, and Eqns. (4.12) and (4.13) pairwise for maximizations, we can move through the graph in a manner similar to Eq. (4.1) and compute approximations μ_n and σ_n^2 to the first two moments of the longest path distribution; since it is assumed to be roughly normal, this suffices to describe the entire distribution. This method appears to have first been proposed for estimating the completion time of PERT networks by Sculli [112], although to simplify the problem he assumed that all of the paths were independent—i.e., that all of the correlation coefficients $\rho_{\epsilon\eta}$ in Eq. (4.11) were zero.

In practice, the moment estimates obtained using Sculli's method tend to be broadly accurate [34]. Moreover, the method is typically much faster than alternatives. However, ignoring the correlations between the maximands is not ideal since common ancestors make path length RVs dependent, even when the weights themselves are not.

Including correlations. Efficiently computing the correlation coefficients between path lengths is tricky. However, Canon and Jeannot [34] proposed two different heuristics which alternatively prioritize precision and speed. The first is a dynamic programming algorithm called Cordyn which recursively computes the correlations using formulae originally derived by Clark [38] for the correlation coefficients between any normal RV τ and a summation or maximization of two other normal RVs ϵ and η,

$$\rho_{\tau,\text{sum}(\epsilon,\eta)} = \frac{\sigma_\epsilon\rho_{\tau\epsilon} + \sigma_\eta\rho_{\tau\eta}}{\sigma_{\text{sum}}} \quad \text{and} \quad \rho_{\tau,\max(\epsilon,\eta)} = \frac{\sigma_\epsilon\rho_{\tau\epsilon}\Phi(\beta) + \sigma_\eta\rho_{\tau\eta}\Phi(-\beta)}{\sigma_{\max}}. \tag{4.14}$$

Cordyn has time complexity $O(ne) \approx O(n^3)$, so is more expensive than Sculli's method, which is quadratic in n. However, numerical experiments performed by Canon and Jeannot suggest that it is almost always more accurate.

In an effort to marry the speed of Sculli's method and the accuracy of Cordyn, Canon and Jeannot proposed an alternative heuristic called CorLCA. The main idea is to construct a simplified version of the DAG called a *correlation tree* that has all the same nodes as the original but only retains a subset of the edges. In particular, where multiple edges are incident to a node—i.e., a maximization must be performed—only the edge expected to contribute most to the maximization is retained in the correlation tree. The motivation here is that the correlation coefficient between any two longest path estimates L_i and L_k can be efficiently approximated by finding the *lowest common ancestor* (LCA) t_a of the corresponding nodes t_i and t_k in the correlation tree: since $L_i \approx L_a + \eta$ and $L_k \approx L_a + \epsilon$ where η and ϵ are independent RVs representing the sums of the costs along the paths between t_a and t_i (resp. t_a and t_k) in the correlation tree, we have

$$\rho_{L_i,L_k} \approx \frac{\sigma_{L_a}^2}{\sigma_{L_i}\sigma_{L_k}}.$$

For every edge, we need to do a lowest common ancestor query, so the time complexity of CorLCA depends to a large extent on the cost of these. Based on similar results in the literature, Canon and Jeannot hypothesize this can be done in linear time, giving an overall time complexity $O(e) \approx O(n^2)$ for the entire algorithm. At any rate, a simulated comparison of several heuristics for approximating the longest path distribution performed by Canon and Jeannot suggested that CorLCA is more efficient than Cordyn with only a relatively small reduction in accuracy [34]. It should however also be noted that it can do badly when longest path length estimates for two or more incident edges at a node are similar since only one of the edges will be retained in the correlation tree.

The canonical method. Another heuristic for estimating the longest path distribution that does not ignore the path length correlations comes from the field of digital circuit design. In the so-called *canonical model* [136], [145], all weight and path length RVs are expressed in the form

$$\mu + \sum_{i=1}^{n} v_i \sigma_i,$$

where μ is the expected value, the v_i are scalar coefficients, and the σ_i are independent unit normal RVs that characterize the variance. The advantage of this is that evaluating summations and maximizations becomes much more straightforward. Let

$$\eta = \mu_\eta + \sum_i v_{\eta,i}\sigma_i \quad \text{and} \quad \epsilon = \mu_\epsilon + \sum_i v_{\epsilon,i}\sigma_i.$$

Then

$$\eta + \epsilon = (\mu_\eta + \mu_\epsilon) + \sum_i (v_{\eta,i} + v_{\epsilon,i})\delta_i.$$

The maximization is only slightly more complex. Let β be as defined in Eq. (4.11). Note that computing β requires the linear correlation coefficient $\rho_{\eta\epsilon}$, which can be efficiently calculated through

$$\rho_{\eta\epsilon} = \frac{\sum_i v_{\eta,i} v_{\epsilon,i}}{\sqrt{\sum_i v_{\eta,i}^2}\sqrt{\sum_i v_{\epsilon,i}^2}}.$$

By definition, we then have

$$\begin{aligned}
\max(\eta,\epsilon) &= \mathbb{P}[\eta > \epsilon]\eta + \mathbb{P}[\epsilon > \eta]\epsilon \\
&= \Phi(\beta)\eta + \Phi(-\beta)\epsilon \\
&= \Phi(\beta)\mu_\epsilon + \Phi(-\beta)\mu_\epsilon + \sum_i \big(\Phi(\beta) v_{\eta,i} + \Phi(-\beta) v_{\epsilon,i}\big)\sigma_i,
\end{aligned}$$

which permits the efficient computation of the maximum. Observe that this is both similar and in some sense contrary to the Clark equation approach, in that the latter precisely computes the first two moments of the maximization of two normal RVs, whereas the canonical method approximates the distribution of the maximization of any two RVs using linear combinations of normal RVs. In their empirical comparison, Canon and Jeannot found that the canonical method tended to fall between Sculli's heuristic and CorLCA in terms of both speed and approximation quality [34].

4.3.3 Fitting a distribution

Canon and Jeannot [33] tested the hypothesis that the longest path distribution will be normal for a large collection of randomly generated graphs. They found that it was broadly reasonable, although there were also a number of violations, most notably when the graphs weights followed exponential distributions (rather than normal or beta, which were the alternatives considered). Analysis suggests that the normality assumption may also be less robust for graphs that are highly parallel or large and dense. To reduce the problem to first principles, the longest path through a graph is exactly that: the longest of all paths. Let P denote the set of all paths through the graph G_π and, for any given path $\delta \in P$, let $|\delta|$ denote its length. Then we have

$$L_n = \max_{\delta \in P} |\delta| \tag{4.15}$$

where each maximand is an RV representing the length of an individual path. When the number of potential critical paths is small, the maximization in Eq. (4.15) is likewise dominated by a small number of terms, each of which can be assumed to be normal by the CLT, so the error in approximating the longest path distribution with a normal is relatively small [120]. However if there are many possible critical paths—which is often the case for large or highly parallel graphs—the error may be much greater. More accurately determining the distribution the longest path follows in such cases would therefore be eminently useful.

Extreme value theory. Dodin and Sirvanci [49] suggest that an *extreme value* distribution—specifically a Gumbel distribution—may be a sensible choice. Although extreme value theory typically concerns itself with the maximum (or minimum) of independent and identically distributed (IID) RVs, which the path lengths generally are not, when paths are loosely correlated we can make an intuitive argument that a Gumbel distribution may be a better fit for the longest path distribution than a normal. Assuming that this is the case, Dodin and Sirvanci also showed how its distribution function, mean and variance could all be approximated. Key to their analysis is the concept of what they call a *dominating path*, intuitively defined as a path that is likely to be critical. Leaving the very important questions of how this should be both formally defined and efficiently computed aside for the moment, suppose that there are d dominating paths. Let δ^* be the path with the greatest mean length μ^* and let σ^* be the standard deviation of its length. The general form of the Gumbel distribution function is

$$G(t) = \exp[-e^{-b(t-a)}],$$

where a is the location parameter and b the scale. By manipulating formulae from Cramer [42], Dodin and Sirvanci show that these parameters can be approximated by

$$a_d = \mu^* + \sigma^* \left(\sqrt{2 \ln d} - \frac{\ln \ln d + \ln 4\pi}{2\sqrt{2 \ln d}} \right)$$

and

$$b_d = \sqrt{2 \ln d} / \sigma^*,$$

respectively. The mean $\mathbb{E}[L_n]$ and variance $\text{Var}[L_n]$ of the longest path distribution are then approximated by

$$\mathbb{E}[L_n] \approx a_d + \gamma / b_d,$$

where γ is the Euler-Mascheroni constant ($\gamma \approx 0.577$), and

$$\text{Var}[L_n] \approx \pi^2 / (6 b_d^2).$$

From a practical perspective, the rub with this approach is that we need to estimate the number of dominating paths, however we choose to define them. In a more recent publication, Dodin [46] suggests the following simple heuristic procedure: consider paths in descending order of their mean length and add them to the set of dominating paths until the difference between the current path length mean and μ^* (as defined above) exceeds $\max\{0.05\mu^*, 0.2\sigma^*\}$—i.e., the path is unlikely to be longer than δ^*. Of course, this in turn begs the question of how we efficiently generate the paths; we will cover very similar theoretical ground in Section 4.4.1 so will defer more detailed treatment of this topic until then, but it is likely to be expensive at any rate.

Mixture of distributions. Instead of a normal or extreme value distribution, Mehrotra, Chai and Pillutla [86] suggest that the longest path is typically more accurately described by a mixture of distributions. Moreover, they also proposed an analytical method for approximating this mixture. The basic idea is to first compute the paths with the greatest expected length in the traditional CPM manner, then treat the tasks which are shared between two or more such paths separately from the rest, so that the maximization over the path length RVs is effectively split into dependent and independent parts. Simulation results for a collection of small benchmark graphs from the literature are presented which indicate that their method is more accurate than the normal or extreme value hypotheses but, to reduce the computational effort, these simulations assumed that all graph weights were IID RVs. Indeed, the big problem with this approach in general is that it is likely to be prohibitively expensive for even moderately sized graphs.

Working backward. Rather than approaching the problem analytically, Salas-Morera et al. [109] took a different tack: using simulation, they analyzed the longest path distribution for a large collection of benchmark graphs, assuming that the weights followed various distributions (beta, normal, triangular and uniform), in order to determine which, if any, standard distribution was the best fit. They found that neither the normal or Gumbel distributions were especially good fits and, in general, a gamma distribution was more accurate. Moreover, they performed a linear regression in order to determine how the first two moments of such a gamma distribution can be best approximated as a function of graph attributes. Ultimately they concluded that

$$\mathbb{E}[L_n] \approx 0.9126u_n + 0.775\sigma_{\min} + 2.9658\ln(D),$$

when the weight distributions were beta, triangular or uniform, and

$$\mathbb{E}[L_n] \approx 1.1336u_n - 0.9153\sigma_{\min} + 1.0927\ln(D),$$

when they were normal, where:

- u_n is the traditional CPM bound on the mean as defined by Eq. (4.2)—i.e., the greatest individual path length mean;
- D is the number of dominating paths, as defined by Dodin [46];

- $\sigma_{\min}$ is the smallest path length variance of all paths in D.

Furthermore, for all weight distributions, the variance was approximated by

$$\mathrm{Var}[L_n] \approx 0.9115\sigma_{\min}.$$

Salas-Morera et al. verified this empirically-derived model on another benchmark graph set, finding that it was again more likely to better represent the longest path distribution than either the normal or Gumbel distributions.

4.4 REDUCE PATHS, THEN MAXIMIZE

In theory, computing the longest path through a DAG with stochastic weights is straightforward: we just calculate the maximization in Eq. (4.15). Of course, leaving aside the question of how we actually do the computation, the issue with this way of framing the problem is that P is usually impractically large, being bounded above only by 2^{n-2} (the number of paths through a fully-connected DAG of size n). It is rare that we can even efficiently enumerate all of the possible paths, which is why the dynamic programming approach of Eq. (4.1) is typically preferred even for scalar weights. But if we could somehow reduce the size of the set of paths which need to be considered, this alternative conception may be more useful. In particular, we suggest that the following broad heuristic framework may be useful for approximating the longest path distribution:

1. Identify a set Q of paths which are good candidates to be the longest.
2. Approximate the distribution of their maximization.

We refer to this framework as *Reduce Paths, then Maximize* (RPM). This is not an entirely new approach: a similar method was used by Ludwig, Möhring and Stork [78] to compute heuristic bounds on the longest path distribution function. However, we will treat the idea in much greater detail than was done there, with several alternatives for each phase considered.

4.4.1 Identifying path candidates

Suppose that we want to identify some subset $Q \subset P$ such that $|Q| \ll |P|$ and $\max_{\delta \in Q} |\delta| \approx \max_{\delta \in P} |\delta|$. Intuitively, perhaps the most natural way to define such a Q is in terms of *path* *criticality*, the probability that a given path will be the longest, i.e.,

$$C(\delta) := \mathbb{P}(|\delta| \geq \max_{\gamma \in P \setminus \{\delta\}} |\gamma|). \tag{4.16}$$

Ideally, we should like to define Q as the set of all paths that have nonzero criticality, or, if this is still impractically large, at least choose Q such that it contains those paths with the greatest criticality. Unfortunately, even computing the single path most likely to be critical is a difficult problem. The main issue is that, unlike for scalar weights, the problem does not have optimal substructure so dynamic programming cannot be applied, as pointed out by Sourash [122], who proposed an alternative algorithm based on mathematical programming. Although runtimes were reported to be reasonable in simulations, his algorithm has exponential worst-case time complexity. Furthermore, it isn't clear how this method can be extended to obtain a set of paths with high criticality, rather than a single path. Heuristics for estimating the most critical paths are therefore typically preferred in practice.

Monte Carlo. The most straightforward way to identify paths with high criticality is to use the MC method. In particular, for a small number of samples, we could simply define Q as the set of those paths which we observed to be critical. Of course, the tricky part here is ensuring that this method gives a more accurate estimate of the longest path distribution than simply using the empirical MC distribution. The ideal scenario would be a graph with a single path which is always critical; assuming we can accurately estimate its length, we could therefore fully describe the longest path distribution after a single sample. More generally, one hopes that there is a "sweet spot" range of sample numbers for which this method is both more accurate than the initial small-scale MC and cheaper than taking enough additional samples such that the empirical solution is superior. Note that more sophisticated ways to estimate path criticalities using MC have also been proposed; see, for example, Sigal, Pritsker and Solberg [116]. However, such methods typically require expensive analysis of the graph before doing the MC, which can make them impractical.

Scalarization. Although it seems sensible that Q should contain the paths which are most likely to be longest, we don't need to estimate path criticalities in order to make a good approximation. For example, like for Dodin's [46] dominating paths, we could define Q as the set of $K = |Q|$ paths with the greatest expected length. The advantage of this approach is that mean path lengths can easily be computed by summing the means of the weights along them. Therefore we can effectively scalarize the weights of the graph, so that the problem becomes: how do we compute the K longest paths through a DAG with scalar weights? This problem is not entirely trivial but it is simpler than the corresponding problem for stochastic weights and can be solved by dynamic programming. The basic idea is to move forward through the DAG and maintain an ordered list of the K longest paths up to the current task. The only tricky part is when the task has $y \geq 2$ parents, since we need to concatenate each of the parent lists to form a new list with yK paths and sort it in order to identify the K longest.

Stochastic dominance. By exploiting the concept of *stochastic dominance*, as defined in Section 4.2.2, Dodin [48], proposed a polynomial time heuristic for approximating the K most critical paths based on a discretization of the (continuous) weights. However, the algorithm requires that the weights follow symmetric distributions and, although polynomial, its $O(n^4)$ complexity may still be prohibitive. Therefore we propose the following alternative heuristic for computing a set Q of K paths which are intuitively the most likely to be critical. Given any two paths δ_1 and δ_2 whose lengths are assumed to be normally distributed by the CLT, i.e., $|\delta_1| \sim \mathrm{N}(\mu_1, \sigma_1^2)$ and $|\delta_2| \sim \mathrm{N}(\mu_2, \sigma_2^2)$, the probability that $|\delta_1| > |\delta_2|$ is given by

$$\mathbb{P}\big(|\delta_1| > |\delta_2|\big) = \Phi\left(\frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2 - \rho_{12}\sigma_1\sigma_2}}\right), \tag{4.17}$$

where ρ_{12} is the linear correlation coefficient between $|\delta_1|$ and $|\delta_2|$. One immediate observation is that if $\mu_1 > \mu_2$ then $\mathbb{P}\big(|\delta_1| > |\delta_2|\big) > 0.5$, so the path with the greatest mean length is likely longer than any other given path through the graph (although not necessarily the maximum of all other paths). For all $i = 1, \ldots, n$, define δ_i^* to be the path from the source to t_i with the greatest expected length. For any given path δ_i which terminates at t_i, define $y(\delta_i) := \mathbb{P}(|\delta_i| > |\delta_i^*|)$—i.e., the probability that path δ_i will be longer than δ_i^*. Ideally, we should like to define Q_i as the K paths which have the largest y values—and therefore the greatest probability of exceeding δ_i^*, where the probabilities are computed using Eq. (4.17). However, this requires the correlation coefficients, which are straightforward but potentially expensive to compute (see next section). We instead define a surrogate function

$x(\delta_i)$ through

$$x(\delta_i) = \frac{\mu_i - \mu_i^*}{\sqrt{\sigma_i^2 + (\sigma_i^*)^2}},$$

where we have assumed $|\delta_i^*| \sim \mathrm{N}(\mu_i^*, (\sigma_i^*)^2)$ and $|\delta_i| \sim \mathrm{N}(\mu_i, \sigma_i^2)$, and also define Q_i to be the set of the K paths terminating at t_i with the greatest x values. Since Φ is an increasing function, intuitively Q_i should then contain most of the paths which are likely to exceed δ_i^* in length (also including δ_i^* itself), so that $Q_n := Q$ contains most of the paths which are likely to be critical. Computing the Q_i can be done in a similar manner as for scalar weighted graphs: at each task, we retain only the K paths with the greatest x values and, when a task has multiple parents, we concatenate the parent path lists and sort the paths therein according to their x values in order to identify the K which should be kept. A complete description of the procedure is given in Algorithm 4.1. (Note that we use the notation $\delta + (t_1, t_2) + t_2$ to indicate that we extend the path δ by adding the edge (t_1, t_2) and task t_2.)

It should be emphasized that this is a heuristic procedure and may not successfully identify the K paths which have the highest criticality. Furthermore, the algorithm may be expensive. The complexity depends on the length of the lists that we have to sort—and therefore the number of parents that each task possesses. In the worst case, this is $O(n)$ on average, so that we effectively have to sort lists of $O(Kn)$ elements at every step. However, the practical cost for realistic schedule graphs may be less drastic than this analysis suggests; we will evaluate this in Section 4.5.4.

4.4.2 Approximating the maximum

Once we have computed the set Q of longest path candidates, we need to approximate $\max_{\delta \in Q} |\delta|$. First, however, we should answer the following question: given a path, how do we estimate its length? Without making any assumptions about the weight distributions, the simplest way is to invoke the CLT: if their lengths are normally distributed, then means and variances are given by summing the means and variances of the constituent weight distributions (recall that we assume that the weights are independent). The problem then reduces to that of approximating the maximum of a set of dependent, differently distributed (DDD) normal RVs. Below, we discuss several different ways this can be done.

Algorithm 4.1: Computing a set of K paths which are among the most likely to be critical.

```
for $i = 2, \dots, n$ do
    Compute $\delta_i^*$ using, e.g., Eq. (4.2)
end
$Q_1 = \{t_1\}$
for $i = 2, \dots, n$ do
    if $\Gamma_i^- = \{t_h\}$ then
        $Q_i = \{\delta + (t_h, t_i) + t_i \mid \delta \in Q_h\}$
    else
        $X_i = \bigcup_{h \in \Gamma_i^-} \{\delta_h + (t_h, t_i) + t_i \mid \delta_h \in Q_h\}$
        Compute $x(\delta_i)$ for all $\delta_i \in X_i$
        Sort $X_i$ in descending order of $x$ values
        $Q_i = X_i[: K]$
    end
end
$Q = Q_n$
```

Monte Carlo. Perhaps the most fitting approach is to use MC simulation—i.e., repeatedly realize the path length RVs according to their distributions and calculate their empirical maxima. The only issue here is that path lengths are typically dependent because of shared task and edge weights. However, note that we can easily estimate the covariance $\text{cov}\left(|\delta_i|, |\delta_j|\right)$ between the lengths of any two paths δ_i and δ_j, where $|\delta_i| \sim \text{N}(\mu_i, \sigma_i^2)$ and $|\delta_j| \sim \text{N}(\mu_j, \sigma_j^2)$, as the sum of the shared weight variances. The traditional way to generate sets of DDD normal path length RVs is then as follows [129]:

1. Construct a covariance matrix Σ such that $\Sigma_{ij} = \text{cov}\left(|\delta_i|, |\delta_j|\right)$ by considering all pairs of path length RVs;
2. Compute the Cholesky factorization $C^T C = \Sigma$ of the covariance matrix;
3. Generate standard (independent) multivariate normal vectors of length $|Q|$;
4. Multiply each of these by the Cholesky factor C;
5. Add the vector of path length means to each.

In our own numerical experiments, we used the `NumPy multivariate_normal` function [93], which efficiently implements the above method, with users only needing to input the covariance matrix and the path length means. Assuming that Q is reasonably-sized, the final four steps of the procedure will be relatively cheap since they largely depend on

$|Q|$ (and generating standard normal RVs can be done efficiently even if a large number of realizations are desired). But computing the covariance matrix can be expensive; in the worst case, all paths may comprise $O(n)$ edges, so that checking shared weights for all $O(|Q|^2)$ pairs may be onerous. Nonetheless, we will investigate the efficiency of this method empirically in Section 4.5.4.

Clark's equations. Rather than using MC, we could in theory use Clark's equations pairwise to approximate the maximum of Q. There are several different ways to do this which are in some sense analogues to the heuristics discussed in Section 4.3.2. For example, ignoring all correlations between the paths would be similar to Sculli's method, whereas using the current most dominant path to estimate the correlations in the intermediate maximizations would be a counterpart to CorLCA. Whichever flavor we use, the main problem with this alternative approach is that we would again have to assume the distribution of the maximization is roughly normal, which, as we shall see in Section 4.5.2, may not always be the case.

Bounds. Although much less studied than the IID case, bounds on the maximum of DDD normal RVs have been proven. For example, Ross gives a general upper bound for the expected value of a set of (not necessarily normal) DDD RVs, which he later specializes for the normal case, as well as mathematical programs for computing tight upper and lower bounds on the expectation in the latter case [104]. None of these are likely to be practical for the full set of paths—i.e., Eq. (4.15)—but they may be applicable for the maximization over Q since it is (hopefully) much smaller. Of course, any such bounds established for the smaller maximization would not technically hold for the full problem, but they may still be useful. Indeed, Ludwig, Möhring and Stork [78] followed precisely this approach, using a modified version of Dodin's algorithm for computing the K most critical paths (see above), in conjunction with the CLT, to obtain heuristic bounds on the distribution function.

As an aside, it is interesting to note that when all weights are normal—so all path lengths are exactly normal as well—there is at least one bound we can prove even for the full set of paths. Since the mean of a given path length is just the sum of the means of the weights along it, we can easily find $\max_{\delta \in P}\left(\mathbb{E}[|\delta|]\right)$ via dynamic programming. This is useful because by Jensen's inequality we have $\mathbb{E}[\max|\delta|] \geq \max\left(\mathbb{E}[|\delta|]\right)$—i.e., the classic

CPM lower bound on the expected value of the longest path. (Of course, this can be proven via other methods to be true even when all weights are not normally distributed.) Similarly, $\max(\text{Var}[|\delta|])$ can be computed cheaply for normal weights, which is Kamburowski's conjectured upper bound on the variance of the longest path in that case. The bound therefore holds if and only if

$$\text{Var}[\max(X_1, \dots, X_m)] \leq \max(\text{Var}[X_1], \dots, \text{Var}[X_m])$$

holds for any set of DDD normal RVs $X_1, \dots, X_m$. While simple counterexamples to this conjecture can be found for RVs from *arbitrary* distributions, as far as we know it remains an open question for normal RVs.

4.5 SIMULATION RESULTS

In order to study the stochastic longest path problem we created a simple software package that facilitates the analysis of DAGs with stochastic weights. As ever, this can be found at the Github repository associated with this thesis[4]. More established software along these lines already exists, such as the *Emapse* package from Canon and Jeannot [34]. However, we decided to create our own, both as a learning exercise and for ease of integration with software used elsewhere in this thesis. Our study comprises three strands. First, in Section 4.5.2, we study empirical longest path distributions. Then, in Section 4.5.3 we compare several heuristics and bounds from the literature. Finally, in Section 4.5.4, we evaluate multiple new heuristics based on the RPM framework proposed in the previous section.

4.5.1 Graphs

As in previous chapters, we used two different sets of graphs in our investigation, the first based on Cholesky factorization and the second with randomly generated topologies from the STG [128]. However, unlike before, these were intended to be *schedule* graphs rather than *task* graphs so there are significant differences, which are elucidated below.

Cholesky DAGs. The topologies of the DAGs in this set are largely similar to the Cholesky factorization task graphs used in previous chapters. However, these graphs represent

[4]https://github.com/mcsweeney90/thesis-code

schedules for such task graphs on accelerated—i.e., CPU and GPU only—target platforms such as those considered in Chapter 2. This means that they have additional disjunctive edges that indicate the order in which processors must execute the tasks. Moreover, since the schedule determines which processor each task is executed on, the node and edge weights of the schedule graph are modeled as RVs with means and variances given by the relevant sample means and variances observed in the BLAS kernel benchmarking that was described in Section 2.5.1 (and summarized in Table 2.3). Similarly, the edge weights represent the communication costs dictated by the schedule and are therefore either scalar zero, if the connected tasks were scheduled on the same processor (or two different CPUs), or an RV with the sample mean and variance of the data movement cost that we measured in our benchmarking experiments. (The latter were not presented in Table 2.3 but were broadly similar to the task execution timing data in terms of their coefficients of variation). Note that at this stage we make no assumptions about the distributions that the weight RVs follow, only that their means and variances are known.

Recall from Chapter 2 that we considered 10 Cholesky factorization task graphs corresponding to matrix tilings from 5×5 (denoted by $N = 5$) to 50×50 ($N = 50$), leading to DAGs with between 35 and 22100 tasks. Furthermore, we considered two different tile sizes ($nb = 128$ and $nb = 1024$) and two different target platforms, defined by the number of GPUs $s \in \{1, 4\}$ since the number of CPUs was $r = 32$ in both cases. For each combination of these parameters, we computed the HEFT [131] schedule for the resulting task graph and target platform, and then constructed the corresponding schedule graph topology by adding a disjunctive edge between a given task and the task immediately scheduled before it on the same processor if no edge already existed between the two. Weights of the graph were then determined by the schedule assignments as described above.

Our Cholesky graph set comprised $10 \times 2 \times 2 = 40$ DAGs in total, corresponding to the possible combinations of N, s and nb. When presenting results we will often split the set into four subsets of 10 graphs, corresponding to combinations of the parameters s and nb.

Randomly generated DAGs. Large-scale empirical investigations along the lines of our study have been done before with randomly generated graphs; see, for example, Canon and Jeannot [34] or Ludwig, Möhring and Stork [78]. Therefore we decided it was best to use only a small set as a point of comparison for the Cholesky graphs, which should be accorded more importance because they are based on a real application. As in previous chapters, we used the topologies of graphs from the STG [128] benchmark as the basis for this set. We used only the 180 graphs with $n = 100$ tasks since the effect of varying size should be apparent from the Cholesky set. To set the weights of the graphs we used the following simple procedure, which takes a parameter μ_v that represents the (mean) coefficient of variation of the weight distributions:

1. Sample the means of all task and edge weight distributions uniformly at random from the interval $[1, 100]$.
2. For each weight, sample the coefficient of variation from a gamma distribution with mean μ_v and standard deviation $0.1\mu_v$ (as done in [34] and with the same justification).
3. Set the variance of each weight as indicated by the sampled coefficient of variation.

One can certainly question whether this procedure yields graphs which accurately reflect those which are likely to arise in a scheduling context. This is somewhat deliberate, as we wished to contrast the performance of various heuristics for these graphs and the more realistic schedule graphs contained in the Cholesky set. Note that, as in [34], we considered $\mu_v \in [0.01, 0.03, 0.1, 0.3]$ and, for each choice and each DAG topology, we repeated the weight-setting procedure 10 times. This means that our STG set therefore comprised $4 \times 180 \times 10 = 7200$ DAGs.

4.5.2 Analysis of longest path distributions

Thus far we have assumed that the mean and variances of all graph weights are known but not the distributions that they follow. In this section, we analyze longest path distributions corresponding to different weight distributions. Ultimately, there are two questions that we wish to answer:

1. How similar is the longest path distribution for different weight distributions?
2. Does the longest path appear to follow any of the standard probability distributions—e.g., normal, gamma, Gumbel—that have been suggested in the literature?

These questions are of obvious practical importance when approximating the longest path distribution. For example, since they are agnostic of the weight distributions, the CLT-based heuristics described in Section 4.3.2 implicitly assume that the first two moments of the longest path distribution will be similar no matter which distributions the weight follow, with the longest path itself assumed to be, at least roughly, normally distributed. Likewise, although no assumptions are made about the form of the longest path distribution itself, the RPM heuristic proposed in Section 4.4 assumes that individual path lengths will be similar for different weight distributions, so long as the weight means and variances do not change.

Generating the distributions. Since computing the true longest path distributions is impractical for the graphs in our test sets, we generated empirical longest path distributions using the MC method with 100,000 samples. This is likely to give highly accurate approximations of the true distributions (see below), although it should of course be noted that they are ultimately still only approximations. For convenience, we assumed that all weights followed the same kind of distribution and considered the following choices:

- Normal, as a best case for the CLT approach since all path lengths are then also exactly normal;
- Gamma, as it often used to model task durations in scheduling problems;
- Uniform, as it differs more broadly from the above than they do from one another.

Whichever distribution the weights followed, the means and variances remained the same, so that realizations were generated in the MC method by sampling from a distribution of the chosen type with the specified mean and variance.

Since we assume that all weights are non-negative because they represent time, if a negative value did occur for the normal or uniform distributions we took its absolute value, so that the distributions in that case were not, in fact, truly normal or uniform. However, this was never relevant for the Cholesky graphs, since the weight coefficients of variation were relatively small, and extremely rare even for the STG set with the largest mean coefficient of variation. Therefore, for convenience, we will continue to refer to those distribution types without specifying this again.

Table 4.1: Time required to generate empirical longest path distributions for Cholesky graphs with different choices of weight distributions. Recall that n is the size of the graph, N is the number of tiles along both axes of the matrix, and nb is the tile size.

		$nb = 128$			$nb = 1024$		
N	n	Normal	Gamma	Uniform	Normal	Gamma	Uniform
5	35	0.2s	0.3s	0.1s	0.1s	0.2s	< 0.1s
10	220	1.1s	1.6s	0.4s	0.9s	1.3s	0.4s
15	680	3.4s	5.0s	1.3s	3.3s	4.9s	1.2s
20	1540	8.2s	11.8s	3.0s	8.1s	11.9s	3.0s
25	2925	15.5s	22.6s	5.9s	21.0s	30.9s	7.4s
30	4960	28.2s	41.2s	10.6s	38.3s	57.5s	13.7s
35	7770	47.4s	1m 7s	17.3s	1m 5s	1m 38s	23s
40	11480	1m 7s	1m 38s	26s	1m 35s	2m 20s	33s
45	16215	1m 35s	2m 22s	37s	2m 11s	3m 18s	46s
50	22100	2m 10s	3m 12s	50s	3m 1s	4m 30s	1m 4s

We have largely avoided discussion of runtimes thus far in this thesis because our `Python` code is not—and is not intended to be—optimal. However, this is an area in which trade-offs between efficiency and accuracy typically need to be made, so some baselines will be necessary going forward. Therefore in Table 4.1 we state how long it took to generate the empirical longest path distributions for the Cholesky graphs with different weight distributions; the data presented are for $s = 1$ only as trends were similar for $s = 4$. (Recall that, unlike in previous chapters, different choices for the parameters s and nb may correspond to different schedule graph topologies, so different runtimes may be expected). There will be some discussion about how these runtimes compare to certain heuristics later.

Metrics. To quantify the similarity between one empirical longest path distribution E_1 and another E_2 corresponding to a different choice of weight distribution, we typically first compared their respective means and variances as a rough guide. Then, as a more rigorous metric, we used the *Kolmogorov-Smirnov (KS) statistic* D, a standard statistical measure that quantifies the difference between F_1, the empirical distribution function of E_1, and its counterpart F_2 through

$$D = \sup_x |F_1(x) - F_2(x)|,$$

i.e., the maximum distance between the two. Note that $D \in [0,1]$ and greater values indicate a greater distance between the two functions. There are many similar metrics that could have been used instead, but the KS statistic is intuitively straightforward to interpret and Canon and Jeannot [34] showed that it is strongly correlated with many other commonly-used metrics. Moreover, following the analysis of Van Slyke [133], we can quantify the accuracy of an empirical longest path distribution compared to the true longest path distribution in terms of the KS statistic. For example, with 100,000 samples we can say, with 95% probability, that the maximum distance between the two is less than 0.005.

How similar? For the Cholesky graphs, to convey an intuitive understanding of the similarity between the longest path distributions for the different choices of weight distributions we present Table 4.2, which gives some summary statistics in the different cases. The data are for the subset of graphs with $s = 1$ and $nb = 128$ only but the most immediate takeaway was apparent for the other parameter choices as well: the statistics are largely very similar. For the smallest graphs, skewness and especially kurtosis can vary considerably, but this appeared to stabilize as the graphs grew. Kolmogorov-Smirnov (KS) statistics for the empirical distribution functions corresponding to the different weight distributions largely supported this intuition. The greatest differences we observed were for gamma and uniform weights, giving a KS statistic of $D = 0.16$ between the two empirical cdfs in the worst case; however, that was an outlier and the average value was just over 0.01. On the whole, the magnitude of the KS statistics also decreased as the graphs grew larger.

We observed similar behavior for the graphs in the STG set. Again, the greatest differences were for gamma and uniform weights, so the figures that follow are for that case. The empirical means were never more than 0.7% different, and on average much smaller. While the standard deviations differed by about 2.8% on average and 15% in the worst case, the average KS statistic for the two empirical cdfs was less than 0.01 and the worst just under 0.1. Altogether this suggests that the answer to the first question posed at the top of this section is that the longest path distributions are usually very similar, no matter which distributions the weights follow, as long as the weight means and variances are fixed.

Which distribution? As noted previously, large-scale experimental studies have been done before on this topic for randomly generated graphs, so we decided to focus only on

Table 4.2: Summary statistics of empirical longest path distributions for Cholesky graphs with $s = 1$ and $nb = 128$. Trends in the data were similar for other parameter choices.

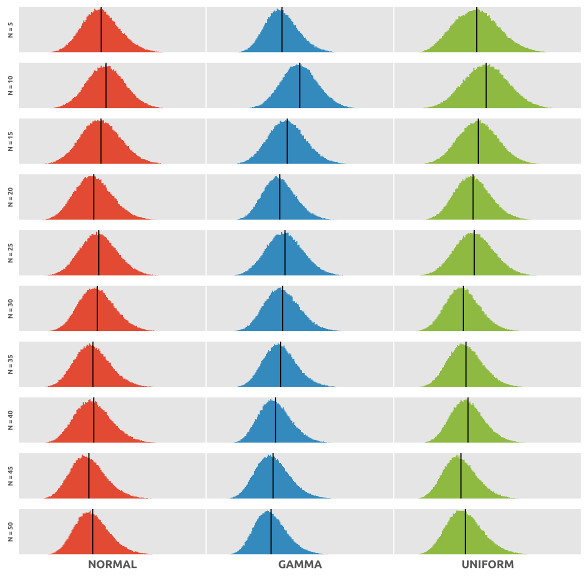
N	n	Weight Dist.	Mean	Median	Std dev.	Skewness	Kurtosis
5	35	Normal	1017.0	1016.4	12.3	0.290	0.172
		Gamma	1017.1	1016.3	12.5	0.380	0.270
		Uniform	1017.2	1016.8	12.3	0.242	−0.144
10	220	Normal	2343.0	2342.7	19.6	0.110	0.017
		Gamma	2342.9	2342.4	19.7	0.160	0.040
		Uniform	2343.2	2343.0	19.6	0.080	−0.083
15	680	Normal	3601.5	3601.0	21.7	0.185	0.081
		Gamma	3602.0	3601.2	22.1	0.241	0.197
		Uniform	3601.8	3601.4	21.4	0.122	−0.011
20	1540	Normal	5863.1	5861.7	31.6	0.297	0.180
		Gamma	5865.3	5863.5	32.4	0.318	0.158
		Uniform	5862.8	5861.4	31.2	0.277	0.127
25	2925	Normal	9073.6	9072.3	40.3	0.194	0.098
		Gamma	9076.4	9075.1	41.0	0.214	0.109
		Uniform	9073.0	9071.8	39.7	0.181	0.111
30	4960	Normal	13574.5	13575.6	45.7	0.352	0.276
		Gamma	13578.4	13575.6	46.9	0.354	0.332
		Uniform	13573.6	13571.3	45.0	0.323	0.260
35	7770	Normal	20196.9	20193.3	58.9	0.381	0.335
		Gamma	20200.1	20196.5	59.7	0.385	0.324
		Uniform	20196.4	20192.9	58.3	0.372	0.317
40	11480	Normal	28835.0	28830.1	72.0	0.406	0.311
		Gamma	28837.9	28832.7	73.1	0.430	0.357
		Uniform	28834.3	28829.7	71.4	0.402	0.360
45	16215	Normal	39165.5	39158.8	85.4	0.487	0.483
		Gamma	39168.2	39161.9	86.0	0.484	0.504
		Uniform	39164.6	39157.8	85.0	0.457	0.380
50	22100	Normal	52224.0	52216.0	100.0	0.497	0.480
		Gamma	52227.6	52219.3	101.0	0.508	0.534
		Uniform	52223.8	52215.9	99.8	0.483	0.471

the Cholesky graphs. Informally, it seems reasonable to surmise that the data presented in Table 4.2 (for those graphs with $s = 1$ and $nb = 128$) are at least roughly normal: the median is very close to the mean and both skewness and (excess) kurtosis are close to zero. This is reinforced by Figure 4.3a, which plots histograms of the data. Inspecting the visualizations, we see that the shapes of the distributions are very similar for all three weight choices, supporting our previous conclusions. Moreover, they all appear to be roughly normal, albeit with a consistent leftward skew for the larger graphs.

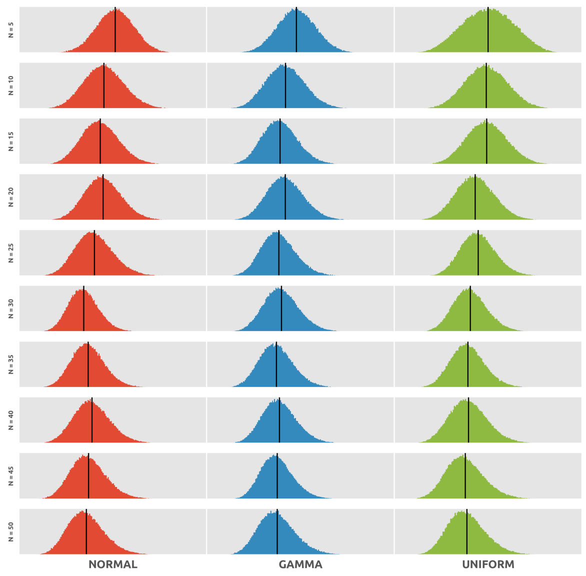
However, when we study Figures 4.3c and 4.3d, which visualize the distributions for the graph subsets with $nb = 1024$, we are forced to draw very different conclusions: many of the distributions are clearly far from normal—or indeed any of the other distributions suggested in the literature. The distributions in question appear bimodal, with spikes to the left of the means. It is not entirely clear what causes this behavior; further investigation, including for applications other than Cholesky factorization, would be useful in the future. Whatever the underlying cause, it is perhaps most striking to observe that for other graphs with the same parameter choices the distributions are still approximately normal. This suggests that the shape of the longest path distribution can differ greatly even for schedule graphs arising from similar task graphs. Moreover, from a practical perspective, this makes it difficult to anticipate when the normality assumption will be reasonable and when it will not be.

4.5.3 Comparison of existing heuristics

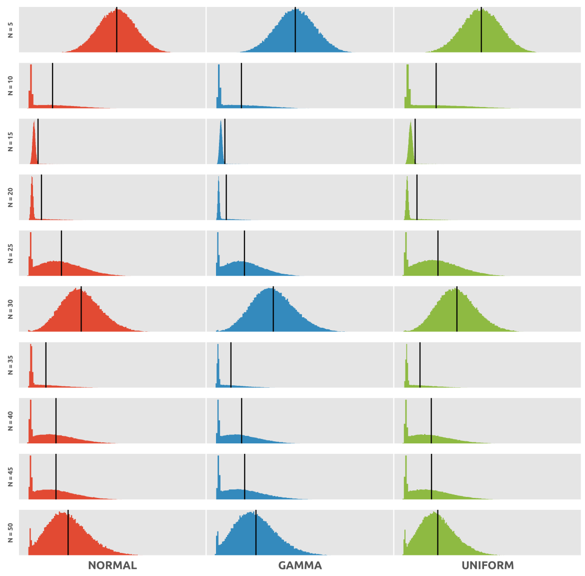
Our experience in the previous section suggests that in general the longest path distribution is only lightly influenced by the distributions that the individual weights follow. On one hand, this supports the use of CLT-based heuristics such as Sculli's method and CorLCA which are not sensitive to the weight distributions. However, note that this also negates one of the traditional drawbacks of the MC method—namely, the need to know all weight distributions exactly—since we can simply sample from any distribution type and get a very close approximation. Furthermore, unlike MC, CLT-based heuristics assume that the longest path distribution will be roughly normal, which our experiments have shown may not always be true—and, perhaps more importantly, that it is difficult to anticipate when this will and will not be the case.



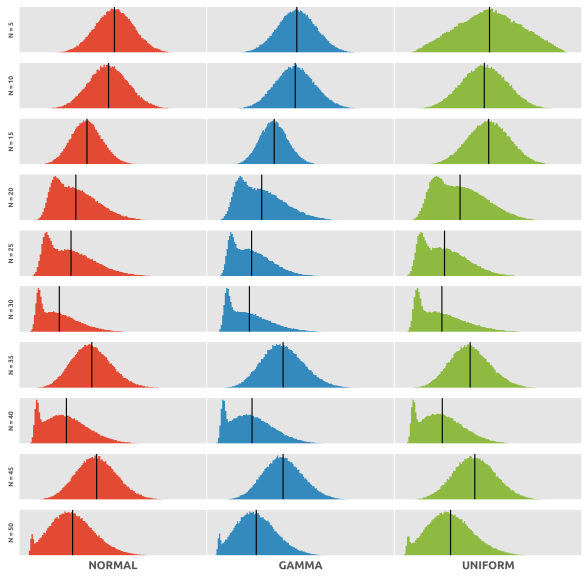
(a) $s = 1$, $nb = 128$.



(b) $s = 4$, $nb = 128$.



(c) $s = 1$, $nb = 1024$.



(d) $s = 4$, $nb = 1024$.

Figure 4.3: Histograms of empirical longest path distributions for Cholesky schedule graphs in the three cases that the weights follow normal, gamma or uniform distributions. Vertical black lines indicate the mean and are included as a visual aid only. Recall that N is the number of tiles along both axes of the matrix, nb is the tile size, and s is the number of GPUs in the target platform.

This would appear to imply that MC is a more fruitful framework for approximating the longest path distribution than CLT-based heuristics. But this neglects the main drawback of MC: namely, the computational effort required, especially when many samples are required. As ever, in practice, a trade-off between accuracy and efficiency is likely to be necessary. Therefore in this section we evaluate several existing heuristics (and bounds) with regard to both the quality of their solutions and how long it took to obtain them. In particular, the methods that we consider here are:

- Sculli's method, as the fastest and most basic example of its type;
- CorLCA, in order to gauge the importance of taking path dependence into account;
- Kamburowski's bounds on the mean and variance, since they have received relatively little attention elsewhere.

First, a warning: results in the previous section suggest that the longest path distribution typically only varies slightly for different weight distributions. But it is not identical either. Assuming that weight distributions are not known exactly, this means that there will always be some "fuzziness" surrounding the actual distribution. Put simply: there is a limit on the possible accuracy of the methods above, since we can't ever know the true solution when the weight distributions aren't known. For the remainder of this section we will assume that the weights actually follow *gamma* distributions and treat the empirical solution obtained via MC with 100,000 samples as the reference longest path distribution. However, this caveat should always be borne in mind. (Note that Kamburowksi's bounds do not technically hold for gamma weights, so in this section we are evaluating them as an approximation only, although they were very rarely violated.)

MC10/100. In Table 4.1 we summarized the runtimes required to obtain empirical distributions for the Cholesky graph set with 100,000 MC samples, but we did not consider a simple question: how many samples do we actually need in order to get a good approximation? Figure 4.4 illustrates the convergence of the MC method for increasing numbers of samples; the data presented are for the subset with $s = 4$, $nb = 1024$ and gamma weights but conclusions were similar in all other cases. Based on the figure, it appears that we typically capture most of the important features of the distribution after around 1000 samples, and often considerably fewer. Therefore, as a comparison for the CLT-based heuristics, we also considered the solutions obtained by the MC method with 10 and 100 samples, which we will refer to as MC10 and MC100 (respectively). Furthermore, to compute MC10/100 we sampled the weights from *uniform* distributions, even though the reference solution is for gamma weights; the idea being that the true distributions may be unknown so, with the conclusions of the previous section in mind, it is sensible to simply use the cheapest distribution type for the MC.

Cholesky graphs. With the exception of Kamburowski's upper bound, which typically overestimated it by around 5–20%, all of the heuristics and bounds gave highly accurate estimates of the longest path mean for the Cholesky graphs, almost always being within 1% of the true solution and often much tighter. Interestingly, this was true even for DAGs whose empirical longest path distributions were far from normal, such as those in Figures 4.3c and 4.3d. This accuracy is impressive, although it is worth noting that even the classic CPM bound was similarly accurate for these graphs. As to the comparative performance, Sculli's method and CorLCA were consistently tighter than Kamburowski's lower bound (in turn always better than the CPM), but MC10 and MC100 were usually superior to both, especially for the larger graphs.

Despite the success for the mean, estimates of the variance were typically far less impressive, as shown in Figure 4.5 (note the logarithmic scale on the y-axes). We see that Kamburowski's bounds, especially the lower, were consistently very loose. CorLCA was clearly superior to Sculli's method, although it became noticeably less accurate as the graphs grew in size. Once again, however, the two MC heuristics were the best overall, with particularly good comparative performance for larger graphs.

Sculli's method and CorLCA compute only the expected value and variance of the longest path distribution, under the assumption that it will be roughly normal, but this did not always appear to be the case for the Cholesky graphs. Formalizing this, we compared the reference empirical distribution functions to normal cdfs with means and standard deviations computed via Sculli's method and CorLCA; Figure 4.6 presents the KS statistics obtained for Sculli's method, CorLCA and MC10/100 (in the latter case, we used the empirical distribution function corresponding to the MC data rather than the normal cdf). Some trends are apparent for all parameter choices; for example, MC10/100 did much better than the CLT-based heuristics for the largest graphs. But there is also some

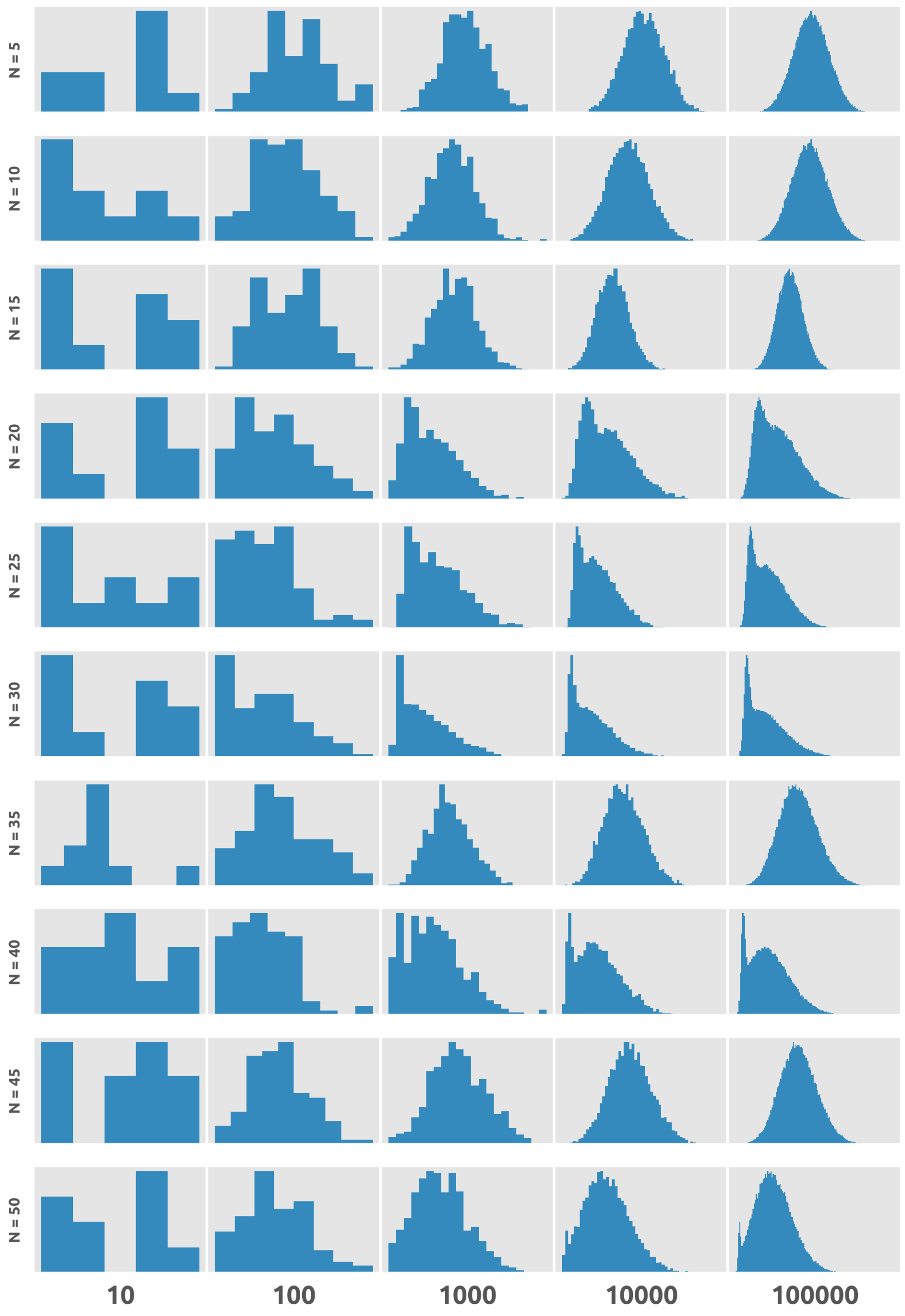
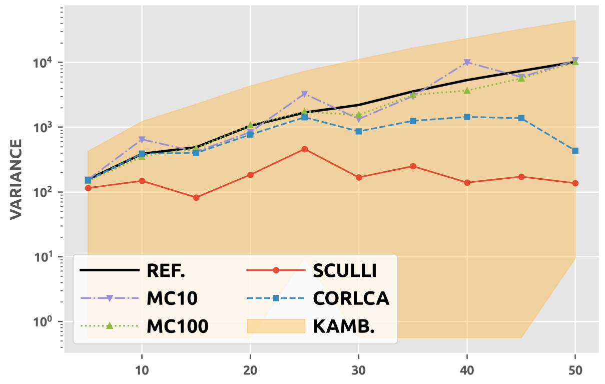
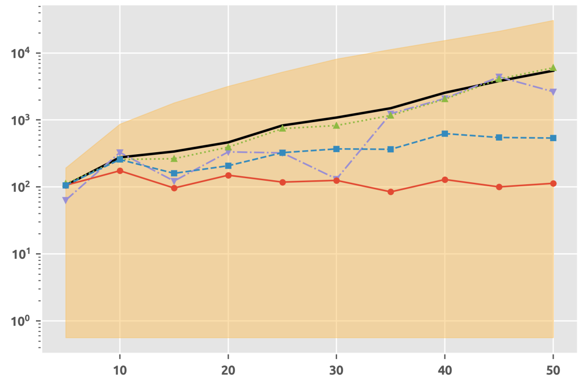


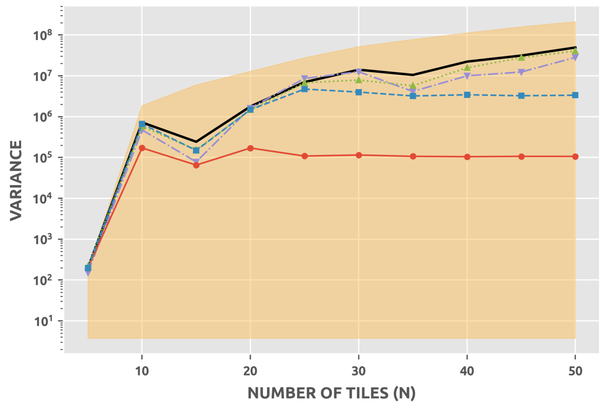
Figure 4.4: Progression of MC solution with increasing numbers of samples for Cholesky graphs with gamma-distributed weights, $s = 4$ and $nb = 1024$. Trends were similar for other weight distributions and choices of s (number of GPUs) and nb (tile size).



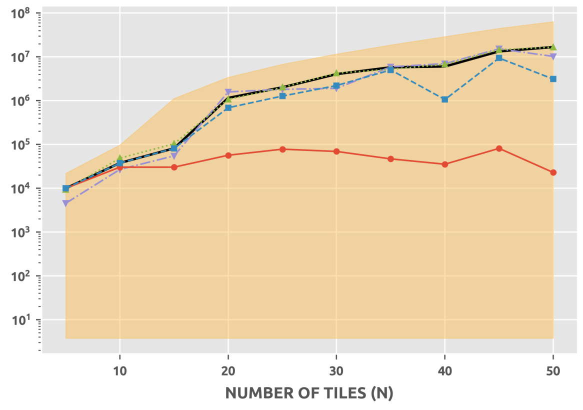
(a) $s = 1$, $nb = 128$.



(b) $s = 4$, $nb = 128$.

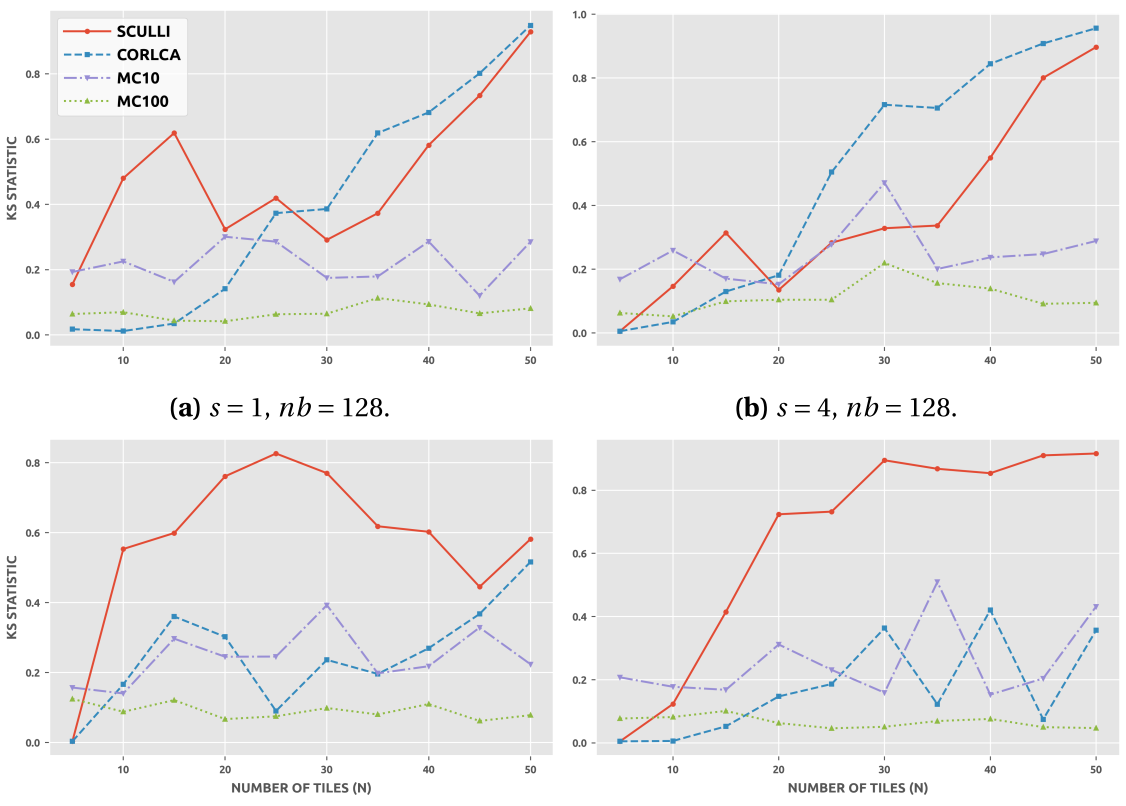


(c) $s = 1$, $nb = 1024$.



(d) $s = 4$, $nb = 1024$.

Figure 4.5: Bounds and approximations to the longest path variance for Cholesky graphs with different combinations of s (number of GPUs) and nb (tile size). Yellow shaded area defines region within Kamburowski's upper and lower bounds. Black curve indicates reference solution. Note the logarithmic scale on the y-axes.



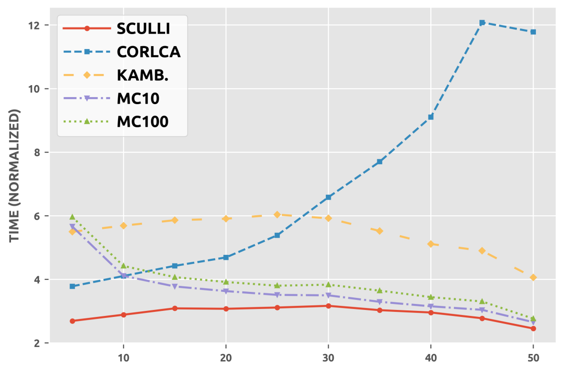
(a) $s = 1$, $nb = 128$. **(b)** $s = 4$, $nb = 128$.

(c) $s = 1$, $nb = 1024$. **(d)** $s = 4$, $nb = 1024$.

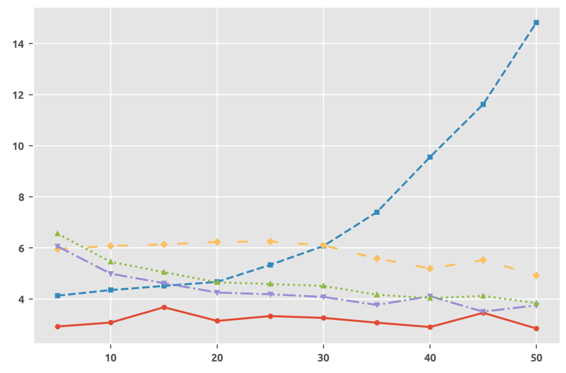
Figure 4.6: Kolmogorov-Smirnov (KS) statistics of Sculli's method, CorLCA and MC heuristics for Cholesky graphs with different combinations of s (number of GPUs) and nb (tile size). Statistics computed though comparison with reference empirical distributions.

notable variation, such as the fact that CorLCA was uniformly better than Sculli's method for $nb = 1024$—more than eight times better for one graph—but they both do about as well as each other for $nb = 128$. We suspect the cause of this disparity lies in the fact that CorLCA is known to perform poorly when there are many paths of similar length; as we will see in the next section, this is the case for both tile sizes but it is far more pronounced for $nb = 128$.

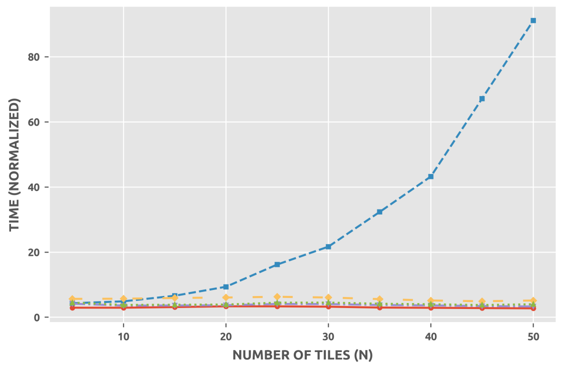
Of course, accuracy is only ever one of the criteria by which we evaluate the quality of heuristic solutions; efficiency is always vitally important as well. Therefore in Figure 4.7 we illustrate the runtimes of our implementations. Note that, for context, the timings are normalized as multiples of the time required to compute the CPM bound (which was always the cheapest). The most notable trend is that CorLCA is initially no more expensive than the others but eventually grows far beyond them, especially for $nb = 1024$. This means that MC10 and MC100 were both cheaper and more accurate for the largest graphs. However, it is difficult to weigh this timing data against the comparative accuracy of the methods



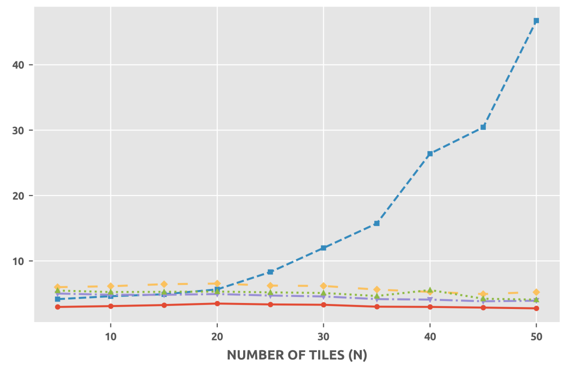
(a) $s = 1$, $nb = 128$.



(b) $s = 4$, $nb = 128$.



(c) $s = 1$, $nb = 1024$.



(d) $s = 4$, $nb = 1024$.

Figure 4.7: Execution time of heuristics and bounds for Cholesky graphs with different combinations of s (number of GPUs) and nb (tile size). Timings are normalized as a multiple of the CPM bound runtime. Kamb. represents time for both Kamburowski's mean and variance bounds.

without making any assumptions about the context in which they are used. Moreover, we emphasize again that these timings are implementation-specific so should be treated with caution. Although not indicated by the figure, since the times are normalized, it should be noted that all of the heuristics and bounds were considerably cheaper than generating the full empirical distribution using 100,000 samples; even CorLCA for the largest graph with $nb = 1024$ was about 20 times faster.

STG set. As with the Cholesky graphs, we found for the STG set that computed approximations of the mean were typically highly accurate for all of the heuristics and bounds. The comparative performance differed slightly, with CorLCA being the best this time, but the average error relative to the reference solution was less than 1% for Sculli's method, CorLCA, MC10/100 and Kamburowksi's lower bound, so all could be considered successful. Estimates of the variance were again much less accurate, however. Figure 4.8 shows the mean percentage error, relative to the reference solution, for the relevant bounds and heuristics, presented according to the mean coefficient of variation of the graph weight RVs μ_v. Note that Kamburowski's lower bound is omitted because it was extremely poor, almost always being zero (and therefore giving an average error > 99%). We see that the performance of the three CLT-based heuristics degraded as μ_v increased, whereas the error for MC10/100 remained roughly the same; both of these trends are perhaps to be expected based on the analysis in earlier sections. Overall, CorLCA was the standout performer, although MC100 was slightly better for $\mu_v = 0.3$. This concurs with Figure 4.9, which shows the mean KS statistics obtained by Sculli's method, CorLCA and MC10/100 (computed in the same manner as for the Cholesky graphs). We see that CorLCA was consistently superior to the others and, on average, was highly accurate.

Comparative runtimes for the heuristics were similar to the trends for the smallest Cholesky graphs indicated by Figure 4.7. MC100 was the most expensive, being about 8.4 times as expensive as the CPM bound on average for the entire set (of 7200 DAGs). The corresponding values for Kamburowski's bounds, Sculli's method, CorLCA and MC10 were 5.5, 2.8, 3.6 and 4.5. Again, it is difficult to weigh accuracy against efficiency without a specific application area in mind, but, given that it was the best-performing and the second cheapest (after Sculli's method, discounting the CPM bound itself), it seems safe to say that CorLCA achieved a good balance between the two objectives for the STG set. The conclusion then would seem to be that it is the best heuristic for these graphs (of those considered). However, there is a big caveat here concerning the MC method: although MC10 and MC100 were both more expensive and less accurate than CorLCA, note that the latter was less than twice as expensive as the former since our code is largely vectorized. Moreover, after several hundred samples, the MC solution typically became more accurate than CorLCA. Therefore a user may decide that, for example, MC1000 better meets their desired efficiency-accuracy trade-off than CorLCA because the runtime increase is not as drastic as one may at first assume.

4.5.4 Evaluating RPM

Our experience so far suggests that, at least for certain graphs, avoiding the normality assumption of the CLT-based heuristics may be wise. The MC method would therefore seem to be the superior approach in such cases. However, those graphs for which we found

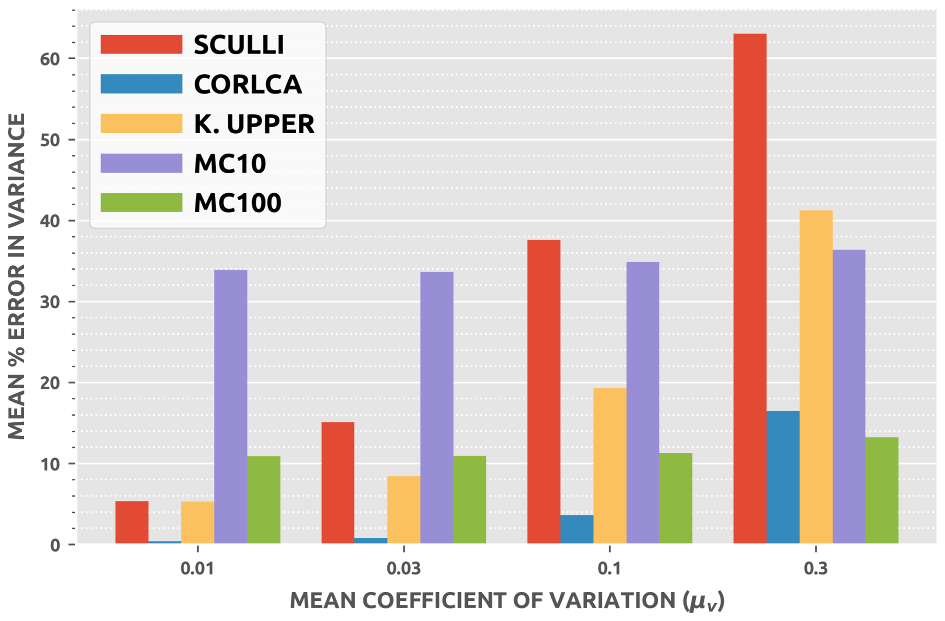


Figure 4.8: Mean percentage error in variance estimates of Sculli's method, CorLCA, Kamburowski's upper bound and MC10/100 for STG set with different mean coefficients of variation μ_v. K. UPPER refers to Kamburowski's upper bound. Note that each bar represents an average over 7200/4 = 1800 graphs.

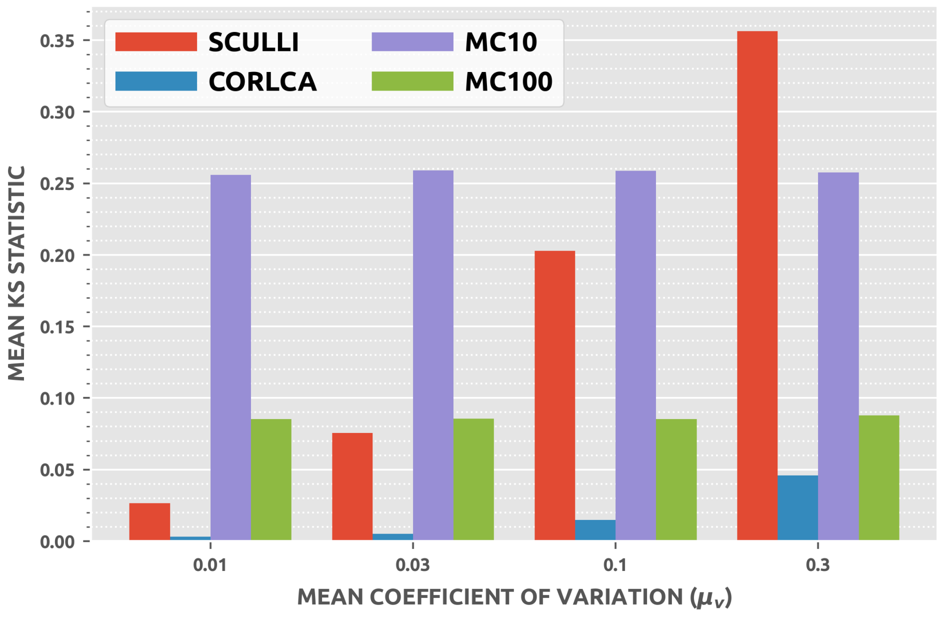


Figure 4.9: Mean Kolmogorov-Smirnov (KS) statistics of Sculli's method, CorLCA and MC10/100 for STG set with different mean coefficients of variation μ_v. Statistics computed though comparison with reference empirical distributions.

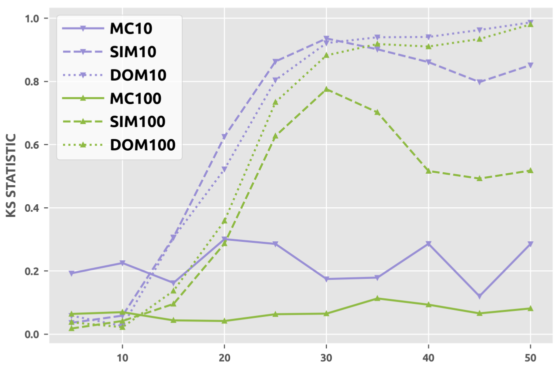
that the CLT-based heuristics were poor—i.e., the large Cholesky graphs—are precisely those for which taking many MC samples is also likely to be expensive. Heuristics which are cheaper than the MC method and do not make assumptions about the longest path distribution would therefore appear to be eminently useful. In this section, we evaluate whether the *Reduce Paths, then Maximize* (RPM) framework that we described in Section 4.4 is capable of achieving this aim.

Variants. The RPM framework is very broad, so we should specify precisely which variants that we considered here. Recall that the heuristic comprises two stages: first, a moderately-sized set Q of candidate longest paths are identified, then their maximum is approximated. For the latter, we always used the MC method based on generating correlated normal RVs that was described in Section 4.4.2. We did 1000 realizations of the path length RVs since they were cheap to generate and this proved more than adequate for convergence; further comments will be made about the runtime implications later. Since the second step is always the same, the RPM variants are therefore defined by how they identify the set of longest path candidates Q. The four methods that we considered here were:

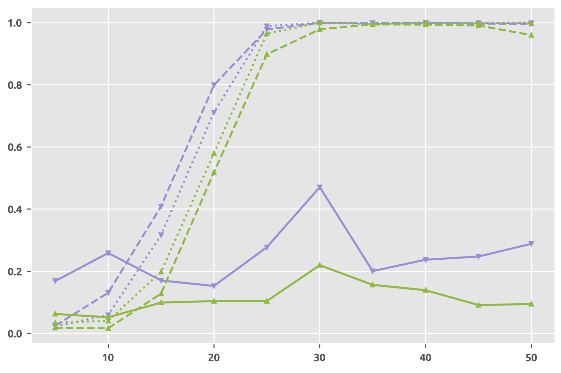
- SIM10 (for *simulation*), defining Q as those paths which were observed to be critical during 10 MC realizations of the graph (with uniform weights);
- SIM100, ditto above but with 100 realizations;
- DOM10 (for *dominance*), computing Q using Algorithm 4.1 with $K = 10$;
- DOM100, ditto above but with $K = 100$.

Note that the size of Q is therefore bounded above by either 10 or 100, depending on the number of MC samples or the value of K in Algorithm 4.1. (Of course, the natural name for SIM10/100 would be *MC*10/100, but we avoid this in order to prevent confusion with the heuristics by those names that have already been defined in this chapter.)

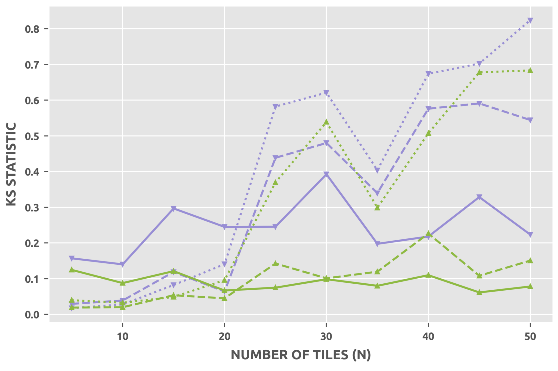
Cholesky graphs. Figure 4.10 shows the Kolmogorov-Smirnov (KS) statistics obtained by the four RPM variants for the Cholesky graph set (where, as before, the MC method with 100,000 samples and gamma distributed weights is used to compute the reference



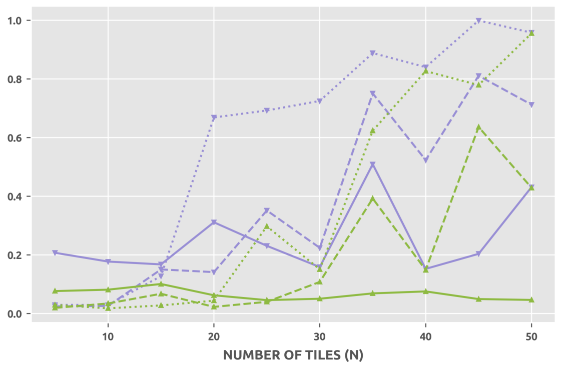
(a) $s = 1$, $nb = 128$.



(b) $s = 4$, $nb = 128$.



(c) $s = 1$, $nb = 1024$.



(d) $s = 4$, $nb = 1024$.

Figure 4.10: Kolmogorov-Smirnov (KS) statistics for RPM variants and MC10/100 for Cholesky graphs with different combinations of s (number of GPUs) and nb (tile size). Statistics computed though comparison with reference empirical distributions.

solution). Included also in the figure are MC10 and MC100, which are as defined in previous sections. We see that none of the RPM variants improved on the corresponding MC heuristics (i.e., with 10 or 100 samples) and were, in fact, almost always considerably worse. As for the CLT-based heuristics, performance degraded as the size of the graphs increased, although to an even more pronounced extent; for $s = 4$ and $nb = 128$, the KS statistics all converged to 1.0 after about $N = 20$.

Although this negative result is obviously unfortunate, it does serve to illustrate certain behavior that may not be apparent for randomly generated graphs. Fundamentally, the problem is the regularity of Cholesky factorization task graphs: there are only four different task types and the same amount of data is transmitted for each task. Combined with the fact that the target platforms for which the schedules were computed comprised multiple identical processors of only two different types, the weights of the schedule graph were therefore relatively homogeneous. This in turn means that there were many paths of similar length, especially for large graphs, and therefore many different paths that could become critical. This is illustrated by Table 4.3, which states the number of different paths that we observed to be critical when generating the empirical longest path distributions in Section 4.5.2 (recall that 100,000 samples were used). For all but the smallest graphs with $nb = 128$ we see that almost every new realization of the weights gave a new critical path. This means that considering any small set of paths is unlikely to be useful, since there are so many paths with similar likelihood of being critical. Even for $nb = 1024$, the number of observed critical paths was far greater than 100 (the maximum size that we allowed for the candidate path set Q).

The obvious solution would seem to be using larger values of K in Algorithm 4.1. The problem is that the computational cost of the algorithm also grows unacceptably high. For the largest graphs, even running Algorithm 4.1 with $K = 100$ took almost as long as generating the reference MC solution (i.e., with gamma weights and 100,000 samples) and it seems likely that very large values of K would be required given the data in Table 4.3. Moreover, constructing the covariance matrix used to generate the correlated path length RVs was even more expensive (although realizing the values afterward was cheap even for large numbers of samples). We experimented with neglecting this step and simply realizing the path lengths as though they were independent but this did not improve performance for the Cholesky graphs and made it worse when RPM does well (see below). Altogether, then, it seems safe to conclude that RPM is ill-suited for the Cholesky graphs.

STG set. The RPM variants were much more accurate for the graphs from the STG set, as shown in Figure 4.11, which presents their mean KS statistics compared to the reference solutions. In particular, we see that they successfully improved on the corresponding MC heuristics. Also included in the figure for reference, however, is CorLCA and we see that it again dominated the comparison, except for $\mu_v = 0.3$ when it was very slightly bettered by DOM100. Moreover, although relatively less so than for the large Cholesky graphs, the RPM variants were again expensive. Compared to CorLCA, on average SIM10 was 1.5 times as expensive; the corresponding values for DOM10, SIM100 and DOM100 were 2.4, 3.3 and 39.2. At this point we should reiterate the warnings that we have proffered throughout about the efficiency of our own implementations, but it is safe to say that the RPM heuristic is unlikely to be a practical alternative to CorLCA unless more efficient

Table 4.3: Number of paths that were observed to be critical for Cholesky graphs (with $s = 1$) using MC method with 100,000 samples and different weight distributions. Results were similar for $s = 4$.

		$nb = 128$			$nb = 1024$		
N	n	Normal	Gamma	Uniform	Normal	Gamma	Uniform
5	35	9	10	6	1	1	1
10	220	92	107	51	7	6	5
15	680	1313	1489	960	67	69	50
20	1540	19452	21140	17860	269	267	240
25	2925	86475	87114	85988	943	979	894
30	4960	91839	92730	91421	1771	1800	1646
35	7770	94318	94946	94302	1131	1141	978
40	11480	89886	90542	89468	6808	7061	6303
45	16215	88399	89334	88181	2538	2635	2381
50	22100	92720	93217	92176	4266	4434	3975

means of identifying longest path candidates and approximating their maximum can be found.

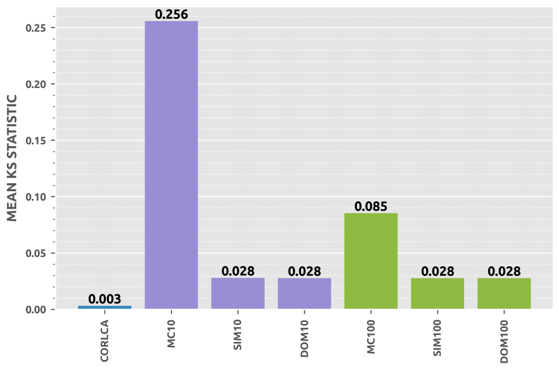
4.6 CONCLUSIONS AND FUTURE WORK

To summarize, we took the following conclusions from the empirical investigation that was described in this chapter.

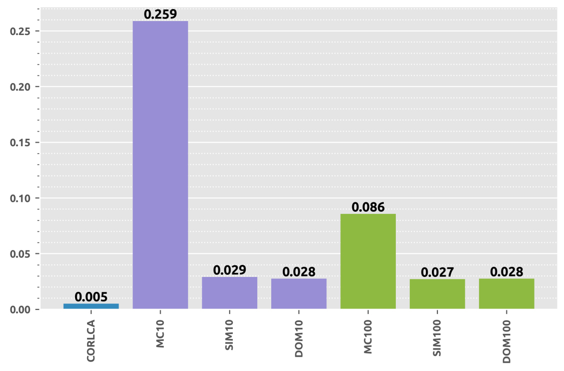
1. Assuming that weight means and variances are fixed, the longest path distribution is usually very similar for different choices of weight distributions. This justifies the use of heuristics such as CorLCA which do not require them but also implies that MC simulation with any choice of weight distributions is likely to be accurate.

2. The assumption that the longest path distribution will be normal, which underlies many existing heuristics, is not always safe. Indeed, we saw that the distribution shape could vary drastically even for DAGs which on the surface may appear to be very similar to one another. Heuristics which do not assume that the longest path distribution will take a certain form would therefore be helpful.

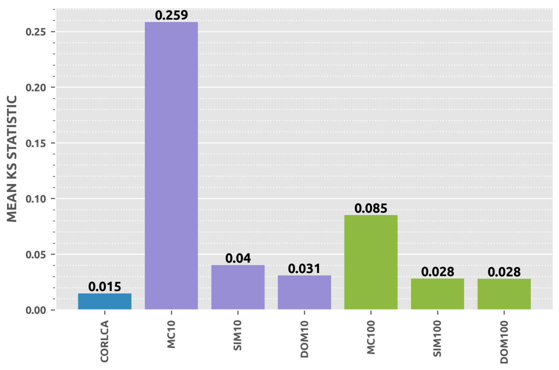
3. CorLCA was the outstanding performer of the CLT-based heuristics. It was far more accurate than Sculli's method for the randomly generated graphs and only slightly



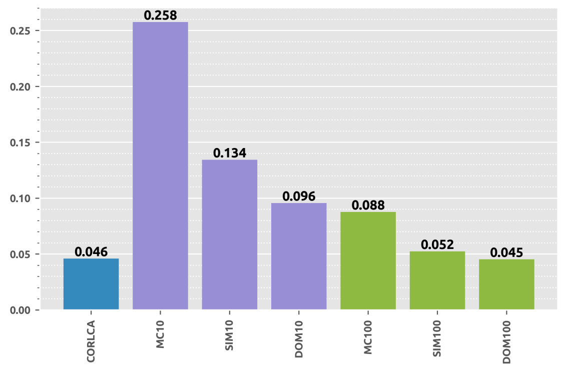
(a) $\mu_v = 0.01$.



(b) $\mu_v = 0.03$.



(c) $\mu_v = 0.1$.



(d) $\mu_v = 0.3$.

Figure 4.11: Mean Kolmogorov-Smirnov (KS) statistics achieved by CorLCA, MC10/100 and RPM variants for the STG set with different mean coefficients of variation μ_v. Statistics computed though comparison with reference empirical distributions.

more expensive in our implementation. Sculli's method was significantly cheaper for the larger Cholesky graphs but both were extremely inaccurate in that case anyway. Kamburowski's bounds appear to be of limited use as heuristics when the weights are not normally distributed since CorLCA in particular was consistently cheaper and more accurate. Furthermore, the lower bound on the variance was always so loose as to be entirely useless.

4. Supporting the first point, we found that the MC10/100 heuristics performed well throughout, even though the weights were sampled from uniform distributions and the reference solutions were for gamma distributed weights. Moreover, they were the only heuristics that were not wholly inaccurate for the large Cholesky graphs. This suggests that a small number of MC samples are often adequate in practice to get good approximate solutions. More broadly, we found that MC was much more efficient compared to alternative heuristics than one might suppose. All of the previous warnings about our implementations should be borne in mind but there is perhaps a salient point here: in general, MC methods are well-suited to modern computer architectures and software.
5. The proposed RPM heuristic framework may be useful in some circumstances but there are practical hurdles that must be overcome in order to make it more widely applicable. The best RPM variants evaluated here were accurate, albeit expensive, for our randomly generated graph set. However, they were completely ineffective for all but the smallest Cholesky graphs since the number of paths that must be considered was very large. Cheaper methods for approximating both the set of longest path candidates and their maximum are therefore clearly needed.

A fundamental issue with all of the heuristics and bounds discussed in this chapter is that they assume the weights of the graph are independent. As we argued earlier, this is unlikely to be entirely correct in scheduling contexts. Efficient new heuristics that do not make this assumption would therefore be of obvious interest in the future.

CHAPTER 5

STOCHASTIC SCHEDULING

In the previous chapter, we studied the problem of predicting the makespan of a schedule when the computation and communication times are not known precisely but instead are expected to follow certain probability distributions. But we did not address a very important related question: how do we actually compute a schedule which is likely to be short given the variation in the schedule costs that we expect to see? In this chapter, we consider this important but difficult *stochastic scheduling problem.* However, the structure differs somewhat from earlier ones in that we do not present a new method but rather describe existing scheduling heuristics, before proposing and evaluating a small modification to one such heuristic.

5.1 A MULTIOBJECTIVE PROBLEM

Suppose that we have a task DAG G and a heterogeneous target platform T. How do we compute a good schedule for G on T? We have already seen many different ways to approach this problem in earlier chapters. But all of these assume that the computation and communication costs are modeled as *scalars*. What should we do if we know that the costs can be more accurately modeled as *random variables* (RVs) instead? Can we use this information to compute a better schedule? Moreover, what does a *better* schedule even mean in this context, given that, if all the costs are stochastic, then the schedule makespan will be as well?

When all schedule costs are scalars, it is clear what an optimal schedule looks like: π^* is optimal if its makespan is minimal over the set of all feasible schedules. There are two issues with extending this definition to the stochastic scheduling problem.

1. As we saw in the previous chapter, computing the makespan distribution of even a single schedule with stochastic costs is an extremely difficult problem, proved to be $\#P$-complete by Hagstrom [60].

2. How do we compare the quality of multiple schedules when their makespans are stochastic?

The first of these means, in particular, that the optimization problem of finding π^* is at least $\#P$-complete itself and therefore finding an optimal solution is likely infeasible, however we define it. Nevertheless, as we also saw in the previous chapter, good heuristics for approximating the makespan distribution exist, so this may not be fatal if we focus only on achieving approximate solutions.

The second issue is more conceptual and there are clearly many different ways that we could compare schedule makespan distributions. Fundamentally, though, a good stochastic schedule should have a high probability of returning a short makespan. In other words: we want it to have a short *expected* length and also want it to be *robust*—resilient to stochasticity in the cost estimates. The obvious metric for the first of these objectives is the expected value of the makespan distribution; clearly, we want this to be as small as possible. However, many metrics have been proposed in the literature for quantifying robustness. Perhaps the most prominent is the *slack*, which has several slightly different definitions but is typically defined for a task as the maximum length of time that its execution can be delayed without increasing the makespan, and for a schedule as the mean of all task slacks [27], [115]. Other examples include *makespan differential entropy* [27], the *miss rate* [115], *stochastic robustness* [113], and many others. Canon and Jeannot [33] compared several common robustness metrics empirically, concluding that the intuitive choice of the *makespan standard deviation* was highly correlated with many of the more complex alternatives (although interestingly not the slack in particular). Given that it is typically straightforward to compute, we will therefore use this as the default metric for quantifying schedule robustness from now on—i.e., we say that one schedule is more robust than another if it has a smaller makespan standard deviation.

Unlike the scalar case, we now have a *multiobjective* optimization problem: we want to minimize both the makespan expected value and its standard deviation. Ideally, of course, we should like a schedule which does both. Unfortunately, such a solution will only exist if both objectives are equivalent, and it should be clear intuitively that minimizing a schedule's length and its robustness are not necessarily the same thing—although, interestingly, it has often been observed empirically that the two *are* correlated to some extent and shorter schedules tend to be more robust than longer ones [33]. (This is perhaps counterintuitive since it is often assumed that good schedules must be in some sense "finely tuned".) Nonetheless, for any multiobjective optimization problem the goal is to find a solution in the *Pareto set*, those solutions which are superior to all others in regards to at least one of the objectives; how we weigh the importance of the two objectives then determines which of these would be viewed as *optimal*. Some stochastic scheduling heuristics explicitly attempt to help users navigate this trade-off, while others focus solely on optimizing only one of the two objectives (usually the expected length).

Note that here, as in the previous chapter, a schedule is assumed to define which tasks each processor should execute *and in what order*. This is sometimes referred to as a *fullahead* schedule [148]. The alternative would be an *assignment* schedule, in which processor selections are respected but the order is decided at runtime depending on when tasks become available for execution. In general, it isn't clear which approach is preferable. Canon et al. [35] studied the problem and found that there was little difference between fullahead and assignment schedules in terms of makespan expected value, although assignment schedules were typically less robust. For the remainder of this chapter, we will assume that all schedules are fullahead schedules, however a systematic investigation of the general problem of how offline scheduling information should actually be used at runtime is one of the areas of future research identified in Chapter 6.

5.2 HEURISTICS

At a high level, heuristics for the stochastic scheduling problem can be divided into two categories: those that convert the problem into a deterministic one in order to compute a schedule, and those that operate directly with the stochastic costs. (Other taxonomies are of course possible.) The first two examples described below are of the first type, whereas the others are examples of the second. Although we describe only four heuristics here, many others exist. Evolutionary algorithms, for example, have frequently been proposed, such as the *Multi-Objective Evolutionary Algorithm* (MOEA) from Canon and Jeannot [33], a refinement of the NSGA-II framework [45]. However, these tend to be expensive: although the MOEA outperformed all alternatives, including HEFT [131], in a simulated comparison, it took on average more than 1000 times as long as HEFT to return a schedule. With the partial exception of *Monte Carlo Scheduling* (see below), which is an iterative method that may be expensive, the heuristics discussed below are highlighted because they are reported to perform well with low time-complexity.

SHEFT. The classic way to convert a stochastic scheduling problem into a deterministic one is to simply replace all of the stochastic costs with their expected values; indeed, this is often implicitly assumed to actually be the case for many deterministic heuristics. However, alternative scalars that more usefully describe the cost distributions could be used instead. An example can be found in the SHEFT (for *stochastic*) heuristic from Tang et al. [126]. SHEFT replaces a generic stochastic schedule cost W, with mean μ and variance σ^2, by the output of a function S, where

$$S(W) = \begin{cases} \mu + \sigma, & \text{if } \sigma \leq \mu, \\ \mu + \mu/\sigma, & \text{otherwise.} \end{cases} \tag{5.1}$$

The heuristic then proceeds exactly as in HEFT [131]. As remarked in Chapter 3, adding the standard deviation to the mean in this way evokes the classic *Upper Confidence Bound* (UCB) rule for multi-armed bandit problems [14] and is done in an implicit attempt to obtain a schedule which minimizes both the makespan mean and standard deviation. Simulations described in the original paper suggest that SHEFT typically obtains schedules which are both shorter and more robust than HEFT; we will evaluate this claim ourselves in Section 5.4.

Of course, although HEFT is used by default, any other deterministic heuristic could be used instead. Indeed, SHEFT is an instance of a more general heuristic framework which comprises the following two steps:

1. All schedule costs are scalarized via some function.
2. A deterministic heuristic is applied to the resulting scalar task graph in order to obtain a schedule.

Clearly there are many possible choices that could be considered within this broad framework.

Monte Carlo Scheduling. Whereas SHEFT scalarizes the costs once in order to obtain a single schedule, *Monte Carlo Scheduling* (MCS) from Zheng and Sakellariou [148] repeats the procedure many times in order to generate a body of potential schedules, with the best among them then chosen. Instead of scalarizing the costs through some function, they are sampled randomly according to their distributions. For each complete sampling of the costs, a deterministic heuristic (e.g., HEFT) is used to compute a schedule, which is added to a set of candidate schedules only if it has not been seen before and passes an inexpensive fitness check. For the latter, Zheng and Sakellariou suggest using the classic CPM bound (see Chapter 4) on the expected value of the makespan; a schedule is retained only if its bound is not much greater than the smallest seen so far. Once the procedure has been repeated a specified number of times, the makespans of the candidate schedules are computed through MC simulation (again, see Chapter 4) and the one with the smallest expected value is selected. Note that, in theory, any other criteria could be used for selection from the candidate schedules instead; for example, we may want the most robust schedule, or that which optimizes some function of the two. In such a case, it would also be sensible to modify the fitness check to reflect the selection criterion as well.

Numerical experiments conducted by Zheng and Sakellariou suggest that, assuming enough realizations of the costs are performed, MCS almost always achieves superior schedules to alternative heuristics. The issue, as with most similar methods that search the solution space, is the computational effort. For each realization of the costs, we need to apply the deterministic heuristic, which is an $O(n^2)$ operation for HEFT in particular and unlikely to be cheaper for any competitive alternative. We also need to approximate the makespan distributions for each of the candidate schedules, which can be expensive, especially if the graph is large, although note that in principle we could use an alternative heuristic rather than MC; our experimental comparison in the previous chapter suggests that CorLCA [34] in particular may be cost effective.

Rob-HEFT. One issue with the scalarization approach of SHEFT described above is that it in some sense accords the mean and standard deviation the same importance, when we may wish to prioritize the expected schedule length over robustness, or vice versa. Another extension of HEFT which permits finer weightings of the two objectives is Rob-HEFT (for *robust*) from Canon and Jeannot [34]. The algorithm takes an *angle* $\alpha \in [0, 90]$ which represents the user's desired trade-off between makespan expected length and robustness; if $\alpha = 0$, only the standard deviation is prioritized and, if $\alpha = 90$, only the expected value. Both phases of the standard HEFT algorithm are then modified in an attempt to obtain the schedule which best meets this specification. In the processor selection phase, the mean and standard deviation of the current schedule makespan distribution are estimated for each of the q processors, assuming that they are chosen for the current task. These makespan approximations can be done using any of the methods discussed in the previous chapter; CorLCA is the suggested default but MC was also considered experimentally. Each of the q makespan estimates gives us a point (mean, standard deviation) in the criteria space. We disregard the dominated points and rescale the others so that they fit in the unit square, then find the one which is closest to the straight line starting from the origin that makes angle α with the x-axis; the task is then scheduled on the corresponding processor. If $\alpha = 0$, the task is scheduled on the processor which minimizes the schedule standard deviation and, if $\alpha = 90$, the expected value. Likewise, in the task prioritization phase, upward ranks are computed for both cost means and standard deviations, then aggregated in a similar manner to compute priorities.

Stochastic Dynamic Level Scheduling. Li et al. [75] proposed a new heuristic called *Stochastic Dynamic Level Scheduling* (SDLS), an extension of the classic deterministic DLS heuristic [117]. The basic idea is to use Clark's equations (4.12) and (4.13) in order to deal with the stochastic costs directly; SDLS therefore assumes that the costs are normally distributed, although, as our experience in the previous chapter suggests, it may still be applicable when this is not the case. First, a counterpart to the upward rank called the *stochastic bottom level* which represents the distribution of the longest path from a given task to the sink is computed for all tasks. This is done using Clark's equations and disregarding path correlations, so is essentially equivalent to Sculli's method (although computed in the opposite direction from the presentation in the previous chapter). Then, in the next phase of the algorithm, SDLS considers the current set of tasks ready for scheduling (initialized with the entry task) and computes for all ready task-processor pairs an RV called the SDL value which is a function of the task's stochastic bottom level. The task-processor pair whose SDL value is *stochastically greater* than the SDL values of all others is then selected, and the indicated task scheduled on the suggested processor; here, an RV is considered to be stochastically greater than another if the 90th percentile of the former is greater than the 90th percentile of the latter.

5.3 ACCELERATING MCS

The MCS heuristic is an iterative method that, given enough time, will almost certainly return a better schedule (however we choose to define it) than the deterministic heuristic that it employs. However, since the computational cost relative to the deterministic heuristic scales with the number of iterations, it is natural to ask if it is possible to accelerate the convergence of the algorithm. In other words: for a given number of schedule production steps, can we increase the quality of the returned schedule?

MCS generates new schedules by scalarizing the costs according to their distributions. An alternative way to scalarize the costs is exemplified by the SHEFT heuristic: a generic RV cost with mean μ and standard deviation σ is scalarized as $\mu + \sigma$, assuming that $\sigma < \mu$, with a small adjustment in the other case. This is claimed to lead to shorter, more robust schedules than using the mean alone. More broadly, we can define a family of possible scalarization functions of the form $S(\mu, \sigma, c) = \mu + c\sigma$, for some scalar parameter c. It is not intuitively clear that the choice $c = 1$ used in SHEFT is any more effective than others. Therefore it may be sensible to consider many different values. In particular, rather than scalarizing schedule costs by sampling from their distributions in MCS, we could apply the S function for different values of c instead. We will refer to this alternative scalarization method as UCB (because of its similarity to the UCB rule for multi-armed bandit problems [14]) and the standard MCS method as MC (for *Monte Carlo*). Although it is difficult to justify why we should expect UCB to converge to a good schedule more quickly than MC, small-scale testing, described below, suggests that this may often be the case.

5.3.1 Simulation environment

To evaluate the proposed optimization of MCS, we used custom simulation software which, as ever, can be found at the Github repository for this thesis[1]. The software effectively

[1]https://github.com/mcsweeney90/thesis-code

combines and extends the two packages that were used in Chapters 3 and 4, the former to simulate the scheduling of task graphs (when costs have been scalarized) and the latter for evaluating the makespan distributions of the resulting schedules. (Recall that we only consider fullahead schedules here, so that computing the makespan distribution of a given schedule with stochastic costs can easily be done by computing the longest path distribution of the corresponding schedule graph, as described in Chapter 4.)

Unlike in previous chapters, we used only a single set of task graphs in our experiments. These were based on the topologies of the 180 DAGs from the STG [128] benchmark with $n = 100$. Note that all cost RVs need only be defined by their means and variances for the scheduling methods that we evaluate here (with the partial exception of MCS; see below). Therefore to generate costs for the task graphs we used the following procedure. First, the expected values of all costs are fixed using the same extension of the *correlation noise-based* (CNB) [32] method that was used for the STG set in Section 3.4.1. Then, for each cost, we determine its variance by randomly sampling its coefficient of variation from a gamma distribution with mean μ_v and standard deviation $0.1\mu_v$, where μ_v is a parameter. As in [34], this is done to avoid bias by ensuring that the standard deviations are not strictly proportional to the means.

Our preliminary testing suggested that there was little difference between the relative performance of MC and UCB when different parameter choices were used in the CNB method to set the cost means (although it may be useful in the future to investigate this further). Therefore, to reduce the runtime, we used only the following set of parameters:

- $q = 4$ (recall that q is the number of processors);
- $r_{\text{task}} = r_{\text{mach}} = 0.5$;
- $V = v_{\text{band}} = v_{\text{ccr}} = 0.5$;
- $\mu_{\text{ccr}} = 1.0$.

These values were chosen randomly. The only parameter that we did vary was μ_v, for which we considered values from the set $\{0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.5\}$. For each choice of μ_v and each DAG topology, we generated 10 sets of costs, so that there were effectively $180 \times 7 \times 10 = 12600$ different graphs.

5.3.2 Methodology

In order to determine which of the two scalarization methods is most effective, for each DAG we did 100 iterations of both. This is a relatively small number compared to the values used for examples in [148] but our exploratory testing with values up to 1000 indicated that the same trends held true. For the MC scalarization method, the costs were sampled from *uniform* distributions with the indicated means and variances; again, preliminary testing suggested there was little difference in performance for other choices (e.g., normal and gamma). For the UCB method, we sampled c uniformly at random from the interval $[0, 3]$ for each of the 100 iterations and then used that value to scalarize the costs. In all cases, we applied HEFT after the costs were scalarized in order to compute a schedule.

Although MCS uses a fitness check to avoid evaluating the makespan distribution of all generated schedules, we did not use that here. Instead, we approximated the makespan distribution of all generated schedules, for both scalarization schemes, using the MC method with 1000 samples and uniform weights. Our investigation in the previous chapter suggests that this is likely to give highly accurate estimates of the makespan distributions, no matter which distributions the costs actually follow, assuming that the means and variances of the cost RVs are fixed. To evaluate the convergence rate of the MC and UCB scalarization methods, we selected the schedule produced by each which minimized the makespan expected value and compared them. We also did likewise for the schedules that maximized another quantity P_h which, for a given schedule π, is intended to roughly approximate the probability that π will be shorter than the corresponding HEFT schedule π_h. This probability was estimated through the following procedure.

1. Compute the empirical makespan distribution of π_h using the MC method (with 1000 samples and uniform weights).

2. Calculate the Mann-Whitney U statistic [82] of the empirical makespan distributions for π and π_h. (Intuitively, U is the number of makespan pairs—one from each empirical distribution—for which π is shorter than π_h.)

3. Divide U by 10^6 (the total number of pairs) to get $P_h \approx \mathbb{P}\left[|\pi| < |\pi_h|\right]$.

In principle P_h is a better measure of schedule quality than the expected value alone, as it implicitly incorporates robustness as well. However, we should emphasize that the procedure used here only yields a crude estimate of the true probability that π will be shorter than π_h and therefore should be treated with a degree of caution.

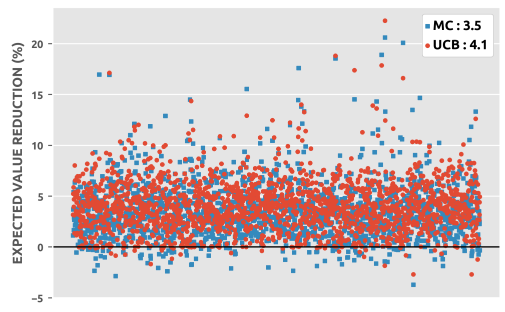
5.4 SIMULATION RESULTS

Figure 5.1 compares the best schedule makespan expected values achieved by the MC and UCB schedules after 100 iterations to the corresponding HEFT schedule makespan mean for different values of μ_v (the mean coefficient of variation in the cost-setting algorithm). We see that UCB is consistently superior to both MC and HEFT. Furthermore, after around $\mu_v = 0.2$ MC fails, on average, to obtain a schedule with a smaller expected makespan than HEFT's, whereas UCB continues to do so for greater values. These conclusions are supported by Figure 5.2, which shows the mean P_h values achieved by MC and UCB. Included also for comparison are the data for SHEFT [126] (see Section 5.2). Again, we see that UCB is superior to MC for all values of μ_v. UCB is also consistently superior to SHEFT, which is in turn likely to return a better schedule than HEFT (i.e., achieve a mean $P_h > 0.5$) for all but $\mu_v = 0.5$.

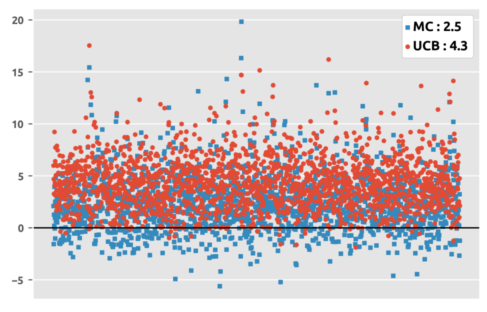
5.5 CONCLUSIONS AND FUTURE WORK

Our simulations suggest that the UCB scalarization method produces good schedules more quickly than MC, especially when the variances of the costs are relatively high. This indicates that UCB may be more effective than MC for producing schedules in the MCS heuristic. We should, however, emphasize again the limitations of the experiments described here, which were for a small set of randomly generated graphs. Extending this investigation to a larger set of graphs, ideally including some from real applications, would clearly be useful in future. Furthermore, it should also be noted that the results presented were only for 100 iterations of each method. Again, small-scale testing indicated that the same trends still held for larger values but this should be established more rigorously; it could well be the case that the performance of UCB begins to stagnate for much greater numbers of iterations.

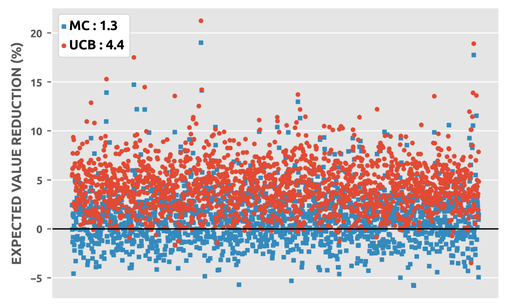
It may also be worthwhile to investigate other methods of scalarizing the costs, although we consider it unlikely that there is any scalarization function likely to lead to



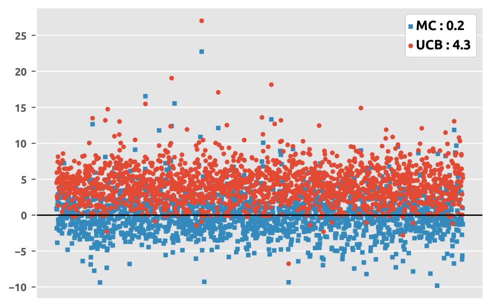
(a) $\mu_v = 0.05$.



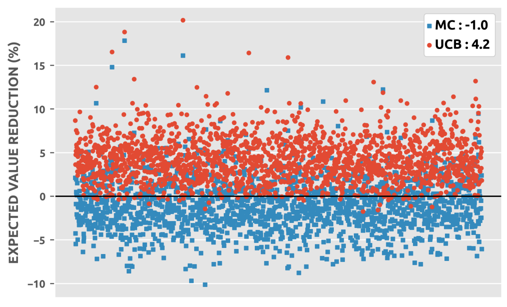
(b) $\mu_v = 0.1$.



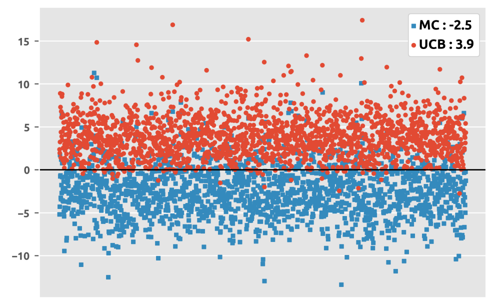
(c) $\mu_v = 0.15$.



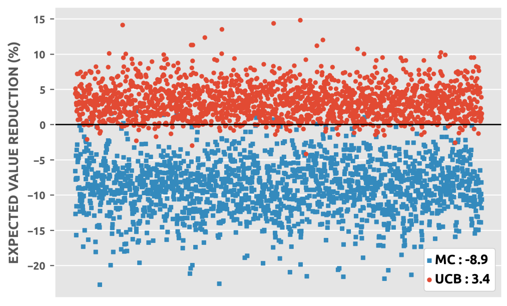
(d) $\mu_v = 0.2$.



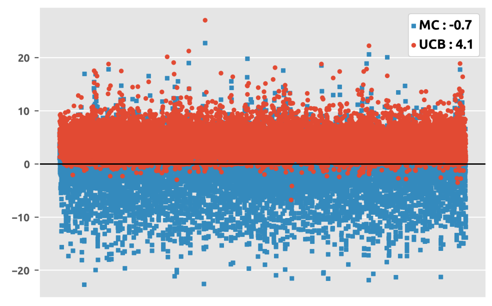
(e) $\mu_v = 0.25$.



(f) $\mu_v = 0.3$.



(g) $\mu_v = 0.5$.



(h) Full set.

Figure 5.1: Percentage reduction in expected value of schedule makespan achieved by the best MC and UCB schedules after 100 iterations compared to the corresponding HEFT schedule. Black horizontal lines indicate zero and are included as a visual aid to help identify when scalarization methods improved on HEFT (i.e., above the line) and when they were worse (below the line). Legends indicate the average percentage reduction in the expected value.

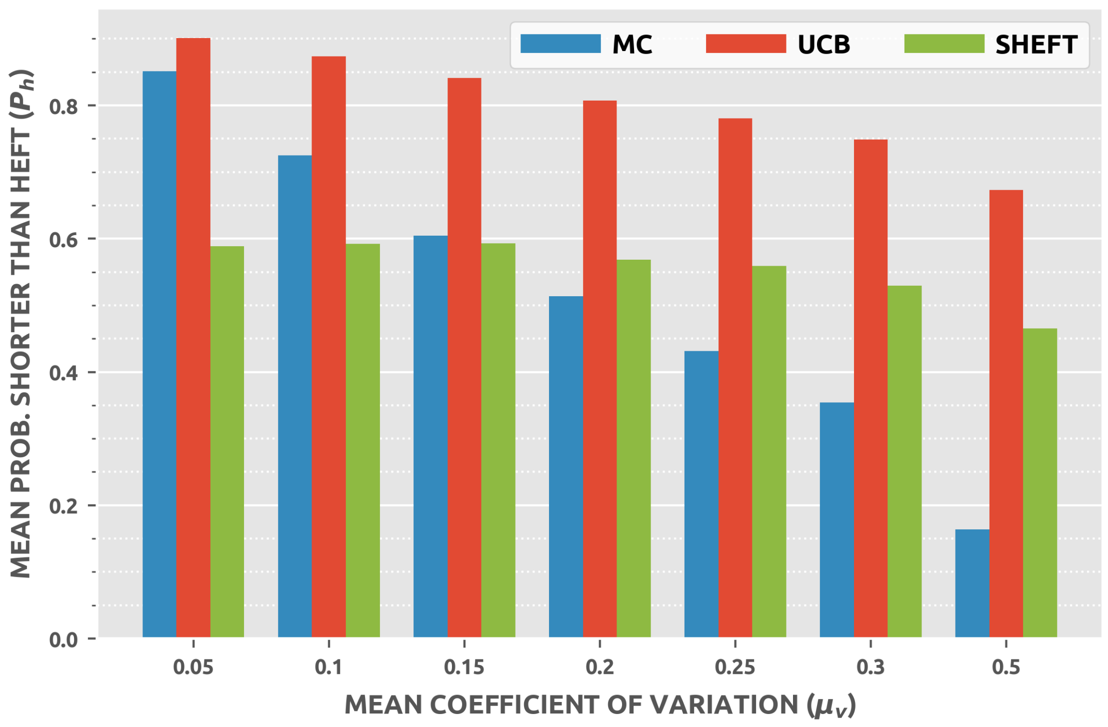


Figure 5.2: Mean P_h values obtained by MC, UCB and SHEFT schedules for different mean coefficients of variation μ_v. Recall that P_h represents an estimate of the probability that a given schedule will be shorter than HEFT's, so that values below 0.5 indicate that the schedule is likely to be worse.

significant improvement. Indeed, we suspect that developing a new scheduling heuristic that operates directly on the stochastic costs when computing a schedule, along the lines of Rob-HEFT and SDLS, is likely to be a much more fruitful subject for future research. As in previous chapters, an exhaustive empirical comparison of multiple task prioritization schemes and processor selection rules would aid the development of such a heuristic.

CHAPTER 6

CONCLUSION

In this thesis we studied several problems related to scheduling in heterogeneous computing. These problems were difficult and subtle. Optimal solutions were practically unobtainable and heuristic algorithms the only realistic choice. We began in Chapter 2 by investigating how the priority-based heuristic framework could be optimized for computing environments with only two different processor types. We proposed several new alternatives for both phases of the framework and compared them experimentally with existing methods, finding that, although there was no universally superior optimization, many of the new methods were effective in certain situations. In Chapter 3 we extended the horizon to generic heterogeneous computing and considered the problem of how critical path estimates can be computed and used in scheduling heuristics. We reviewed existing methods and proposed several new ones, including one approach based on a stochastic interpretation of the problem which consistently bettered alternatives when critical path estimates were used to prioritize tasks. In Chapter 4 we studied the problem of predicting a schedule's makespan when the values that the costs will take at runtime are not known exactly. We analyzed empirical makespan distributions for a real application and observed unexpected behavior that motivated our proposal of a new heuristic framework for tackling the problem. Finally, in Chapter 5, we investigated how schedules could be computed which are robust to uncertainty in their estimates, suggesting an optimization of an existing iterative algorithm that appears to accelerate its convergence.

6.1 FUTURE RESEARCH DIRECTIONS

Based on our experience conducting the research described in this thesis, we suggest that the following may be promising avenues of future work in the area of heterogeneous scheduling.

6.1.1 Hybrid scheduling

This thesis was wholly concerned with *offline* scheduling. Even in Chapters 4 and 5, where we assumed that schedule costs were stochastic, the motivation was always that we either are given, or wished to compute, a static schedule which is to be followed as closely as possible at runtime. Intuitively, what we have called stochastic scheduling here can be viewed as attempting to find the best possible schedule, *assuming that the costs actually follow the distributions that we expect at runtime.* But in reality there is always the potential that our expectations will be violated, whether this is due to modeling inaccuracies or any number of outside factors, such as resource contention and failure. What, then, do we do? Should we continue to follow the schedule we have, or do we need to *reschedule* and modify it somehow?

An extended study focused on the general topic of if, and how, static schedules should be modified as new information becomes available at runtime (sometimes called *hybrid* scheduling [148]) would be of clear interest. It is well-known that, when deviations from the cost estimates are not too extreme, good static schedules are typically superior to online alternatives [2], [33], [107]. Therefore it is sensible to follow a computed static schedule up to the point at which we detect that the estimates used are so inaccurate as to make the schedule detrimental. Key to making this decision is the ability to quickly update estimated schedule makespan distributions at runtime, as realizations of the costs become apparent. This problem is closely related to the subject of Chapter 4 and, of course, the most straightforward solution is to simply divide the schedule graph into two parts, a "realized" part for which the costs have already been incurred and a "future" part for which the costs have not. Any of the standard stochastic longest path heuristics can then be used to evaluate the longest path distribution of the future subgraph—i.e., the expected time remaining if we continue to follow the schedule. The issue with this approach is that it may be expensive, especially if we wish to do it often. An efficient means of updating the estimated schedule makespan without the cost associated with working through the graph would therefore be useful.

More broadly, it would be helpful to establish, through a combination of theoretical analysis and simulated experiments, how we should act if we do decide to deviate from a given static schedule. One extreme here would be to disregard the schedule altogether and employ an online (e.g., greedy) heuristic for the remaining tasks, however it could well be that superior performance can be achieved by only deviating from the static schedule in a few specific ways. Of course, there will be no one-size-fits-all answer to such a general problem, but it would be useful to determine guidelines for different circumstances (e.g., varying levels of uncertainty about future cost estimates) in order to help make this decision.

6.1.2 Reinforcement learning

Many of the scheduling heuristics discussed in this thesis utilize *dynamic programming* (DP) at some point; in HEFT, for example, computing task priorities is done in a classic DP manner. DP is closely related to *reinforcement learning* (RL), a kind of machine learning that has achieved many prominent successes in recent years and therefore become popular for tackling difficult scheduling problems. With this in mind, it may be helpful to consider if RL—and indeed machine learning more broadly—can be successfully applied in this context.

Overview. Reinforcement learning can be considered either as a specific form of supervised learning or as a distinct type of learning in its own right [123]. The key idea is learning through *interaction*. The framework for a reinforcement learning problem is as follows: an *agent* interacts with an *environment* whose dynamics are not entirely known. The agent observes the *state* of the environment and then takes some *action* based on this information for which it receives feedback in the form of some scalar *cost*. The goal of the agent is to minimize the total cost that the agent incurs over all time. (Note that the aim of reinforcement learning is often framed in terms of maximizing some *reward* rather than minimizing costs, but the two are equivalent.) A wide variety of RL algorithms have been proposed such as REINFORCE [139], Monte Carlo Tree Search [41], Sarsa [105] and Q-learning [137], [138] but they all essentially aim to help the agent make this minimization.

RL and scheduling. Reinforcement learning as a concept has been around for decades but was traditionally impractical for interesting problems because it was unable to deal with the large number of possible states and actions that these problems usually possess, called the *curse of dimensionality* [22]. However advances in recent years, particularly in the area of *deep* reinforcement learning—when deep neural networks are used for function approximation—suggest that this may no longer be the case. In particular, the successes achieved by Google's DeepMind in training deep RL agents to play Atari games at superhuman levels of performance [90] and defeat the best human players at the board game Go [118], [119] clearly show that reinforcement learning is now a viable method for dealing with complex problems. Due at least in part to these achievements, reinforcement learning has enjoyed a recent surge in popularity and has been applied to a wide range of problems in many different areas. Scheduling is no exception: RL solutions were considered at least as far back as Zhang and Dietterich [146] but interest has increased sharply in the last few years.

Mao et al. designed DeepRM, a deep reinforcement learning agent intended to handle general dynamic resource management problems in computer systems and networks [83]. Using a variant of the REINFORCE algorithm, they trained a deep neural network to balance multiple different objectives (such as minimizing total job execution time and reducing processor idle times). Simulations suggested that their agent performed well compared to state-of-the-art approaches. A notable limitation of their work is that they did not consider jobs with intra-task dependencies (such as those which can be represented by task graphs). Modifications to the DeepRM agent and its extension to static problems were later investigated by Ye et al. [144].

The use of Q-learning for dynamic load balancing in distributed heterogeneous systems was investigated by Parent et al. [97], and later Samreen and Kumar [110], and Tong et al. [130]. Li et al. [74] used artificial neural networks to estimate task execution times in their extension of the classic *Modified Critical Path* (MCP) scheduling heuristic [140]. Training neural networks to predict thread performance on the diverse processing cores of a heterogeneous system in order to maximize total throughput was also considered in the more recent work of Nemirovsky et al. [92]. Ipek et al. proposed a reinforcement learning based memory controller for multicore processors [63]. Simulations suggested that it was capable of outperforming the existing approaches at that time but the RL-based controller was never actually implemented on an physical chip because of the prohibitive cost of constructing such hardware.

Possible applications. From a scheduling perspective, the most straightforward use of RL would simply be as a means of constructing a near-optimal schedule. By suitably defining the problem in the classic RL framework, it should be possible to train an RL agent for computing a good schedule, however we choose to define *good*. This is likely to be expensive, so RL would fill a similar niche to evolutionary algorithms or other approaches that are used when very high-quality schedules are required. More speculatively, and tying into the previous future research topic, it would be interesting to establish if RL could be used to train a scheduling agent how to act at runtime in response to delays or resource failures.

Bibliography

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson, A comparison of list schedules for parallel processing systems, *Commun. ACM*, vol. 17, no. 12, pp. 685–690, 1974.

[2] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, Are static schedules so bad? A case study on Cholesky factorization, in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 1021–1030.

[3] E. Agullo, O. Beaumont, L. Eyraud-Dubois, *et al.*, Bridging the gap between performance and bounds of Cholesky factorization on heterogeneous platforms, in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 34–45.

[4] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra, Comparative study of one-sided factorizations with multiple software packages on multi-core hardware, in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–12.

[5] E. Agullo, C. Augonnet, J. J. Dongarra, *et al.*, QR factorization on a multicore node enhanced with multiple GPU accelerators, in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, IEEE, 2011, pp. 932–943.

[6] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, Task-based FMM for heterogeneous architectures, *Concurrency and Computation: Practice and Experience*, vol. 28, no. 9, pp. 2608–2629, 2016.

[7] I. Ahmad and Y.-K. Kwok, On exploiting task duplication in parallel program scheduling, *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 9, pp. 872–892, 1998.

[8] M. Ait Aba, L. Zaourar, and A. Munier, Efficient algorithm for scheduling parallel applications on hybrid multicore machines with communications delays and energy constraint, *Concurrency and Computation: Practice and Experience*, vol. 32, no. 15, e5573, 2020.

[9] ——, Polynomial scheduling algorithm for parallel applications on hybrid platforms, in *Combinatorial Optimization*, M. Baïou, B. Gendron, O. Günlük, and A. R. Mahjoub, Eds., Cham: Springer International Publishing, 2020, pp. 143–155.

[10] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram, Generic algorithms for scheduling applications on heterogeneous platforms, *Concurrency and Computation: Practice and Experience*, vol. 31, no. 15, e4647, 2019.

[11] E. Anderson, Z. Bai, C. Bischof, *et al.*, *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[12] H. Arabnejad and J. G. Barbosa, List scheduling algorithm for heterogeneous systems by an optimistic cost table, *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2014.

[13] K. Asanovic, R. Bodik, J. Demmel, *et al.*, A view of the parallel computing landscape, *Commun. ACM*, vol. 52, no. 10, pp. 56–67, 2009.

[14] P. Auer, Using confidence bounds for exploitation-exploration trade-offs, *J. Mach. Learn. Res.*, vol. 3, pp. 397–422, 2003.

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[16] N. Bansal and S. Khot, Optimal long code test with one free bit, in *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, 2009, pp. 453–462.

[17] O. Beaumont, V. Boudet, and Y. Robert, A realistic model and an efficient heuristic for scheduling with heterogeneous processors, in *Proceedings 16th International Parallel and Distributed Processing Symposium*, 2002.

[18] O. BEAUMONT, L. EYRAUD-DUBOIS, and S. KUMAR, Approximation proofs of a fast and efficient list scheduling algorithm for task-based runtime systems on multi-cores and GPUs, in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 768–777.

[19] O. BEAUMONT, L.-C. CANON, L. EYRAUD-DUBOIS, *et al.*, Scheduling on two types of resources: A survey, *ACM Comput. Surv.*, vol. 53, no. 3, 2020.

[20] O. BEAUMONT, L. EYRAUD-DUBOIS, and S. KUMAR, Fast approximation algorithms for task-based runtime systems, *Concurrency and Computation: Practice and Experience*, vol. 30, no. 17, e4502, 2018.

[21] O. BEAUMONT, J. LANGOU, W. QUACH, and A. SHILOVA, A makespan lower bound for the scheduling of the tiled Cholesky factorization based on ALAP schedule, in *EuroPar 2020 - 26th International European Conference on Parallel and Distributed Computing*, ser. Proceedings of EuroPar 2020, Warsaw / Virtual, Poland: Springer, 2020.

[22] R. BELLMAN, *Dynamic Programming*, ser. Dover Books on Computer Science. Dover Publications, 2013.

[23] L. F. BITTENCOURT, R. SAKELLARIOU, and E. R. M. MADEIRA, DAG scheduling using a lookahead variant of the Heterogeneous Earliest Finish Time algorithm, in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 27–34.

[24] D. BLAAUW, K. CHOPRA, A. SRIVASTAVA, and L. SCHEFFER, Statistical timing analysis: From basic principles to state of the art, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 4, pp. 589–607, 2008.

[25] R. BLEUSE, S. HUNOLD, S. KEDAD-SIDHOUM, F. MONNA, G. MOUNIÉ, and D. TRYSTRAM, Scheduling independent moldable tasks on multi-cores with GPUs, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2689–2702, 2017.

[26] R. BLEUSE, S. KEDAD-SIDHOUM, F. MONNA, G. MOUNIÉ, and D. TRYSTRAM, Scheduling independent tasks on multi-cores with GPU accelerators, *Concurrency and Computation: Practice and Experience*, vol. 27, no. 6, pp. 1625–1638, 2015.

[27] L. Böloni and D. C. Marinescu, Robust scheduling of metaprograms, *Journal of Scheduling*, vol. 5, no. 5, pp. 395–412, 2002.

[28] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, PaRSEC: Exploiting heterogeneity to enhance scalability, *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[29] B. Bramas, Impact study of data locality on task-based applications through the Heteroprio scheduler, *PeerJ Computer Science*, vol. 5, e190, 2019.

[30] T. D. Braun, H. J. Siegel, N. Beck, *et al.*, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[31] J. M. Burt and M. B. Garman, Conditional Monte Carlo: A simulation technique for stochastic network analysis, *Management Science*, vol. 18, no. 3, pp. 207–217, 1971.

[32] L.-C. Canon, P.-C. Héam, and L. Philippe, Controlling the correlation of cost matrices to assess scheduling algorithm performance on heterogeneous platforms, *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, e4185, 2017, e4185 cpe.4185.

[33] L.-C. Canon and E. Jeannot, Evaluation and optimization of the robustness of DAG schedules in heterogeneous environments, *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, pp. 532–546, 2010.

[34] L.-C. Canon and E. Jeannot, Correlation-aware heuristics for evaluating the distribution of the longest path length of a DAG with random weights, *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3158–3171, 2016.

[35] L.-C. Canon, E. Jeannot, R. Sakellariou, and W. Zheng, Comparative evaluation of the robustness of DAG scheduling heuristics, in *Grid Computing*, Springer, 2008, pp. 73–84.

[36] L.-C. Canon, L. Marchal, and F. Vivien, Low-cost approximation algorithms for scheduling independent tasks on hybrid platforms, in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds., Cham: Springer International Publishing, 2017, pp. 232–244.

[37] K. Chronaki, A. Rico, M. Casas, *et al.*, Task scheduling techniques for asymmetric multi-core systems, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 2074–2087, 2017.

[38] C. E. Clark, The greatest of a finite set of random variables, *Operations Research*, vol. 9, no. 2, pp. 145–162, 1961.

[39] C. T. Clingen, A modification of Fulkerson's PERT algorithm, *Operations Research*, vol. 12, no. 4, pp. 629–632, 1964.

[40] E. G. Coffman and R. L. Graham, Optimal scheduling for two-processor systems, *Acta informatica*, vol. 1, no. 3, pp. 200–213, 1972.

[41] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in *International conference on computers and games*, Springer, 2006, pp. 72–83.

[42] H. Cramér, *Mathematical Methods of Statistics (PMS-9)*. Princeton University Press, 1999.

[43] L. Dagum and R. Menon, OpenMP: An industry standard API for shared-memory programming, *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[44] J. Darlington, A. J. Field, P. G. Harrison, P. 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*, ser. PARLE '93, London, UK, UK: Springer-Verlag, 1993, pp. 146–160.

[45] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II, in *Parallel Problem Solving from Nature PPSN VI*, M. Schoenauer, K. Deb, G. Rudolph, *et al.*, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 849–858.

[46] B. DODIN, A practical and accurate alternative to PERT, in *Perspectives in Modern Project Scheduling*, J. Józefowska and J. Weglarz, Eds. Boston, MA: Springer US, 2006, pp. 3–23.

[47] ——, Bounding the project completion time distribution in PERT networks, *Operations Research*, vol. 33, no. 4, pp. 862–881, 1985.

[48] ——, Determining the K most critical paths in PERT networks, *Operations Research*, vol. 32, no. 4, pp. 859–877, 1984.

[49] B. DODIN and M. SIRVANCI, Stochastic networks and the extreme value distribution, *Computers & Operations Research*, vol. 17, no. 4, pp. 397–409, 1990.

[50] J. J. DONGARRA, J. DU CROZ, S. HAMMARLING, and I. S. DUFF, A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

[51] A. DURAN, E. AYGUADÉ, R. M. BADIA, *et al.*, OmpSs: A proposal for programming heterogeneous multi-core architectures, *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[52] S. E. ELMAGHRABY, On the expected duration of PERT type networks, *Management Science*, vol. 13, no. 5, pp. 299–306, 1967.

[53] L. EYRAUD-DUBOIS, *Pmtool: Post-mortem analysis tool for StarPU scheduling studies*, [Online; accessed 23-March-2021].

[54] G. S. FISHMAN, Estimating critical path and arc probabilities in stochastic activity networks, *Naval Research Logistics Quarterly*, vol. 32, no. 2, pp. 249–261, 1985.

[55] D. R. FULKERSON, Expected critical path lengths in PERT networks, *Operations Research*, vol. 10, no. 6, pp. 808–817, 1962.

[56] A. GERASOULIS and T. YANG, A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors, *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276–291, 1992.

[57] R. L. GRAHAM, Bounds on multiprocessing timing anomalies, *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[58] R. GRAHAM, E. LAWLER, J. LENSTRA, and A. KAN, Optimization and approximation in deterministic sequencing and scheduling: A survey, in *Discrete Optimization II*, ser. Annals of Discrete Mathematics, P. Hammer, E. Johnson, and B. Korte, Eds., vol. 5, Elsevier, 1979, pp. 287–326.

[59] A. GUPTA, S. IM, R. KRISHNASWAMY, B. MOSELEY, and K. 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, pp. 1242–1253.

[60] J. N. HAGSTROM, Computational complexity of PERT problems, *Networks*, vol. 18, no. 2, pp. 139–147, 1988.

[61] D. S. HOCHBAUM and D. B. SHMOYS, Using dual approximation algorithms for scheduling problems theoretical and practical results, *J. ACM*, vol. 34, no. 1, pp. 144–162, 1987.

[62] INTEL, *Intel Math Kernel Library*, [Online; accessed 21-April-2021].

[63] E. IPEK, O. MUTLU, J. F. MARTÍNEZ, and R. CARUANA, Self-optimizing memory controllers: A reinforcement learning approach, in *2008 International Symposium on Computer Architecture*, 2008, pp. 39–50.

[64] E. JEANNOT, Symbolic mapping and allocation for the Cholesky factorization on NUMA machines: Results and optimizations, *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 283–290, 2013.

[65] J. KAMBUROWSKI, An upper bound on the expected completion time of PERT networks, *European Journal of Operational Research*, vol. 21, no. 2, pp. 206–212, 1985.

[66] ——, Normally distributed activity durations in PERT networks, *Journal of the Operational Research Society*, vol. 36, no. 11, pp. 1051–1057, 1985.

[67] S. KEDAD-SIDHOUM, F. MONNA, and D. TRYSTRAM, Scheduling tasks with precedence constraints on hybrid multi-core machines, in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 27–33.

[68] J. E. KELLEY, Critical-path planning and scheduling: Mathematical basis, *Operations Research*, vol. 9, no. 3, pp. 296–320, 1961.

[69] G. B. Kleindorfer, Bounding distributions for a stochastic acyclic network, *Operations Research*, vol. 19, no. 7, pp. 1586–1601, 1971.

[70] W. H. Kohler, A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems, *IEEE Transactions on Computers*, vol. C-24, no. 12, pp. 1235–1238, 1975.

[71] S. Kumar, Scheduling of dense linear algebra kernels on heterogeneous resources, University of Bordeaux, PhD thesis, 2017.

[72] J. K. Lenstra and A. H. G. Rinnooy Kan, Complexity of scheduling under precedence constraints, *Operations Research*, vol. 26, no. 1, pp. 22–35, 1978.

[73] A. Li, S. L. Song, J. Chen, *et al.*, Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect, *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2020.

[74] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. 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, pp. 89–100.

[75] K. Li, X. Tang, B. Veeravalli, and K. Li, Scheduling precedence constrained stochastic tasks on heterogeneous cluster systems, *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 191–204, 2015.

[76] T. A. Lopes Genez, R. Sakellariou, L. F. Bittencourt, E. R. Mauro Madeira, and T. Braun, Scheduling scientific workflows on clouds using a task duplication approach, in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, 2018, pp. 83–92.

[77] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. J. Dongarra, A scalable high performant Cholesky factorization for multicore with GPU accelerators, in *High Performance Computing for Computational Science – VECPAR 2010*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 93–101.

[78] A. LUDWIG, R. H. MÖHRING, and F. STORK, A computational study on bounding the makespan distribution in stochastic project networks, *Annals of Operations Research*, vol. 102, no. 1-4, pp. 49–64, 2001.

[79] D. G. MALCOLM, J. H. ROSEBOOM, C. E. CLARK, and W. FAZAR, Application of a technique for research and development program evaluation, *Operations Research*, vol. 7, no. 5, pp. 646–669, 1959.

[80] U. MANBER, *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[81] U. OF MANCHESTER, *The Computational Shared Facility 3*, [Online; accessed 20-May-2021].

[82] H. B. MANN and D. R. WHITNEY, On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other, *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[83] H. MAO, M. ALIZADEH, I. MENACHE, and S. KANDULA, Resource management with deep reinforcement learning, in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16, Atlanta, GA, USA: ACM, 2016, pp. 50–56.

[84] J. J. MARTIN, Distribution of the time through a directed, acyclic network, *Operations Research*, vol. 13, no. 1, pp. 46–66, 1965.

[85] T. MCSWEENEY, N. WALTON, and M. ZOUNON, An efficient new static scheduling heuristic for accelerated architectures, in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, *et al.*, Eds., Cham: Springer International Publishing, 2020, pp. 3–16.

[86] K. MEHROTRA, J. CHAI, and S. PILLUTLA, A study of approximating the moments of the job completion time in PERT networks, *Journal of Operations Management*, vol. 14, no. 3, pp. 277–289, 1996.

[87] H. MEUER, E. STROHMAIER, J. J. DONGARRA, H. SIMON, and M. MEUER, *Green500*, [Online; accessed 25-April-2021].

[88] ——, *Top500*, [Online; accessed 25-April-2021].

[89] S. MITTAL and J. S. VETTER, A survey of CPU-GPU heterogeneous computing techniques, *ACM Comput. Surv.*, vol. 47, no. 4, 69:1–69:35, 2015.

[90] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, Human-level control through deep reinforcement learning, *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[91] C. Moler, Matrix computation on distributed memory multiprocessors, *Hypercube Multiprocessors*, vol. 86, no. 181-195, p. 31, 1986.

[92] 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)*, 2017, pp. 121–128.

[93] Numpy, *Multivariate normal*, [Online; accessed 03-November-2021].

[94] NVIDIA, *CuBLAS*, [Online; accessed 21-April-2021].

[95] ——, *CuSOLVER*, [Online; accessed 21-April-2021].

[96] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, GPU computing, *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[97] J. Parent, K. Verbeeck, J. Lemeire, A. Nowe, K. Steenhaut, and E. Dirkx, Adaptive load balancing of parallel applications with multi-agent reinforcement learning on heterogeneous systems, *Scientific Programming*, vol. 12, no. 2, pp. 71–79, 2004.

[98] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, Hierarchical task-based programming with StarSs, *The International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[99] R. Prodan and M. Wieczorek, Bi-criteria scheduling of scientific grid workflows, *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 2, pp. 364–376, 2010.

[100] P. Raghavan and C. D. Tompson, Randomized rounding: A technique for provably good algorithms and algorithmic proofs, *Combinatorica*, vol. 7, no. 4, pp. 365–374, 1987.

[101] V. Rayward-Smith, UET scheduling with unit interprocessor communication delays, *Discrete Applied Mathematics*, vol. 18, no. 1, pp. 55–71, 1987.

[102] P. Robillard and M. Trahan, Technical note—expected completion time in PERT networks, *Operations Research*, vol. 24, no. 1, pp. 177–182, 1976.

[103] M. Rocklin, *BLAS operations counts*, [Online; accessed 20-May-2021].

[104] A. M. Ross, Useful bounds on the expected maximum of correlated normal variables, *Industrial and Systems Engineering*, no. 03W-004, 2003.

[105] G. A. Rummery and M. Niranjan, On-line Q-learning using connectionist systems, Tech. Rep., 1994.

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

[107] R. Sakellariou and H. Zhao, A low-cost rescheduling policy for efficient mapping of workflows on grid systems, *Scientific Programming*, vol. 12, no. 4, pp. 253–262, 2004.

[108] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, Scheduling workflows with budget constraints, in *Integrated Research in GRID Computing: CoreGRID Integration Workshop 2005 (Selected Papers) November 28–30, Pisa, Italy*, S. Gorlatch and M. Danelutto, Eds. Boston, MA: Springer US, 2007, pp. 189–202.

[109] L. Salas-Morera, A. Arauzo-Azofra, L. García-Hernández, J. M. Palomo-Romero, and J. L. Ayuso-Muñoz, New approach to the distribution of project completion time in PERT networks, *Journal of Construction Engineering and Management*, vol. 144, no. 10, p. 04 018 094, 2018.

[110] F. Samreen and M. S. H. Khiyal, Q-learning scheduler and load balancer for heterogeneous systems, *Journal of Applied Sciences*, vol. 7, no. 11, pp. 1504–1510, 2007.

[111] J. Schutten, List scheduling revisited, *Operations Research Letters*, vol. 18, no. 4, pp. 167–170, 1996.

[112] D. Sculli, The completion time of PERT networks, *Journal of the Operational Research Society*, vol. 34, no. 2, pp. 155–158, 1983.

[113] V. Shestak, J. Smith, H. J. Siegel, and A. A. Maciejewski, A stochastic approach to measuring the robustness of resource allocations in distributed systems, in *2006 International Conference on Parallel Processing (ICPP'06)*, 2006, pp. 459–470.

[114] K. R. SHETTI, S. A. FAHMY, and T. BRETSCHNEIDER, Optimization of the HEFT algorithm for a CPU-GPU environment. In *PDCAT*, 2013, pp. 212–218.

[115] Z. SHI, E. JEANNOT, and J. J. DONGARRA, Robust task scheduling in non-deterministic heterogeneous computing systems, in *2006 IEEE International Conference on Cluster Computing*, 2006, pp. 1–10.

[116] C. SIGAL, A. PRITSKER, and J. SOLBERG, The use of cutsets in Monte Carlo analysis of stochastic networks, *Mathematics and Computers in Simulation*, vol. 21, no. 4, pp. 376–384, 1979.

[117] G. C. SIH and E. A. LEE, A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, 1993.

[118] D. SILVER, A. HUANG, C. J. MADDISON, *et al.*, Mastering the game of Go with deep neural networks and tree search, *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[119] D. SILVER, J. SCHRITTWIESER, K. SIMONYAN, *et al.*, Mastering the game of Go without human knowledge, *Nature*, vol. 550, no. 7676, p. 354, 2017.

[120] D. SINHA, H. ZHOU, and N. V. SHENOY, Advances in computation of the maximum of a set of gaussian random variables, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 8, pp. 1522–1533, 2007.

[121] F. SONG, A. YARKHAN, and J. J. DONGARRA, Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems, in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11.

[122] H. M. SOROUSH, The most critical path in a PERT network, *Journal of the Operational Research Society*, vol. 45, no. 3, pp. 287–300, 1994.

[123] R. S. SUTTON and A. G. BARTO, *Reinforcement Learning: An Introduction*, 2nd. MIT press Cambridge, 2017.

[124] O. SVENSSON, Hardness of precedence constrained scheduling on identical machines, *SIAM Journal on Computing*, vol. 40, no. 5, pp. 1258–1274, 2011.

[125] X. TANG, K. LI, R. LI, and B. VEERAVALLI, Reliability-aware scheduling strategy for heterogeneous distributed computing systems, *Journal of Parallel and Distributed Computing*, vol. 70, no. 9, pp. 941–952, 2010.

[126] X. TANG, K. LI, G. LIAO, K. FANG, and F. WU, A stochastic scheduling algorithm for precedence constrained tasks on grid, *Future Generation Computer Systems*, vol. 27, no. 8, pp. 1083–1091, 2011.

[127] S. THIBAULT, On Runtime Systems for Task-based Programming on Heterogeneous Platforms, Habilitation à diriger des recherches, University of Bordeaux, 2018.

[128] T. TOBITA and H. KASAHARA, A standard task graph set for fair evaluation of multiprocessor scheduling algorithms, *Journal of Scheduling*, vol. 5, no. 5, pp. 379–394, 2002.

[129] Y. L. TONG, Statistical computing related to the multivariate normal distribution, in *The Multivariate Normal Distribution*. New York, NY: Springer New York, 1990, pp. 181–201.

[130] Z. TONG, Z. XIAO, K. LI, and K. LI, Proactive scheduling in distributed computing–A reinforcement learning approach, *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2662–2672, 2014, Special Issue on Perspectives on Parallel and Distributed Processing.

[131] H. TOPCUOGLU, S. HARIRI, and M.-Y. WU, Performance-effective and low-complexity task scheduling for heterogeneous computing, *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[132] L. VALIANT, The complexity of computing the permanent, *Theoretical Computer Science*, vol. 8, no. 2, pp. 189–201, 1979.

[133] R. M. VAN SLYKE, Letter to the editor—Monte Carlo methods and the PERT problem, *Operations Research*, vol. 11, no. 5, pp. 839–860, 1963.

[134] A. VASUDEVAN and D. GREGG, Mutual inclusivity of the critical path and its partial schedule on heterogeneous systems, *CoRR*, vol. abs/1701.08800, 2017.

[135] B. VELTMAN, B. LAGEWEG, and J. LENSTRA, Multiprocessor scheduling with communication delays, *Parallel Computing*, vol. 16, no. 2, pp. 173–182, 1990.

[136] C. VISWESWARIAH, K. RAVINDRAN, K. KALAFALA, *et al.*, First-order incremental block-based statistical timing analysis, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 2170–2180, 2006.

[137] C. J. C. H. WATKINS, Learning from delayed rewards, PhD thesis, King's College, Cambridge, 1989.

[138] C. J. C. H. WATKINS and P. DAYAN, Q-learning, *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[139] T. M. WILLIAMS, Criticality in stochastic networks, *Journal of the Operational Research Society*, vol. 43, no. 4, pp. 353–357, 1992.

[140] M.-Y. WU and D. D. GAJSKI, Hypertool: A programming aid for message-passing systems, *IEEE transactions on parallel and distributed systems*, vol. 1, no. 3, pp. 330–343, 1990.

[141] W. WU, A. BOUTEILLER, G. BOSILCA, M. FAVERGE, and J. J. DONGARRA, Hierarchical DAG scheduling for hybrid distributed systems, in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, IEEE, 2015, pp. 156–165.

[142] A. YARKHAN, J. KURZAK, and J. J. DONGARRA, QUARK users' guide: QUeueing And Runtime for Kernels, *University of Tennessee Innovative Computing Laboratory Technical Report*, 2011.

[143] N. YAZIA-PEKERGIN and J.-M. VINCENT, Stochastic bounds on execution times of parallel programs, *IEEE Transactions on Software Engineering*, vol. 17, no. 10, pp. 1005–1012, 1991.

[144] Y. YE, X. REN, J. WANG, *et al.*, A new approach for resource scheduling with deep reinforcement learning, *CoRR*, vol. abs/1806.08122, 2018.

[145] L. ZHANG, W. CHEN, Y. HU, and C. C. CHEN, Statistical static timing analysis with conditional linear MAX/MIN approximation and extended canonical timing model, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 6, pp. 1183–1191, 2006.

[146] W. ZHANG and T. G. DIETTERICH, A reinforcement learning approach to job-shop scheduling, in *IJCAI*, vol. 95, 1995, pp. 1114–1120.

[147] H. Zhao and R. Sakellariou, An experimental investigation into the rank function of the Heterogeneous Earliest Finish Time scheduling algorithm, in *Euro-Par 2003 Parallel Processing*, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 189–194.

[148] W. Zheng and R. Sakellariou, Stochastic DAG scheduling using a Monte Carlo approach, *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1673–1689, 2013.