autotuning with machine learning of OpenMP task applications

Luís Felipe Garlet Milani

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 Mai 2016

Présentée par

Luís Felipe GARLET MILANI

Thèse dirigée par **Jean-François MÉHAUT**
et codirigée par **Lucas MELLO SCHNORR**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Autotuning Assisté par Apprentissage Automatique de Tâches OpenMP

Autotuning with Machine Learning of OpenMP Task Applications

Thèse soutenue publiquement le **11th November 1111**,
devant le jury composé de :

Yves DENNEULIN
Professeur, Université Grenoble Alpes, France, Président
Alfredo GOLDMAN
Professeur, Universidade de São Paulo (USP), Brazil, Rapporteur
Raymond NAMYST
Professeur, Université de Bordeaux, France, Rapporteur
Laércio LIMA PILLA
Chargé de Recherche, CNRS, Université Paris Sud, Examinateur
Jean-François MÉHAUT
Professeur, Université Grenoble Alpes, France, Directeur de thèse
Lucas MELLO SCHNORR
Professeur, Université Grenoble Alpes, France, Co-Directeur de thèse

Abstract

Modern computer architectures are highly complex, requiring great programming effort to obtain all the performance the hardware is capable of delivering. Indeed, while developers know potential optimizations, the only feasible way to tell which of them is faster for some platform is to test it. Furthermore, the many differences between two computer platforms, in the number of cores, cache sizes, interconnect, processor and memory frequencies, etc, makes it very challenging to have the same code perform well over several systems. To extract the most performance, it is often necessary to fine-tune the code for each system. Consequently, developers adopt autotuning to achieve some degree of portable performance. This way, the potential optimizations can be specified once, and, after testing each possibility on a platform, obtain a high-performance version of the code for that particular platform. However, this technique requires tuning each application for each platform it targets. This is not only time consuming but the autotuning and the real execution of the application differ. Differences in the data may trigger different behaviour, or there may be different interactions between the threads in the autotuning and the actual execution. This can lead to suboptimal decisions if the autotuner chooses a version that is optimal for the training but not for the real execution of the application. We propose the use of autotuning for selecting versions of the code relevant for a range of platforms and, during the execution of the application, the runtime system identifies the best version to use using one of three policies we propose: Mean, Upper Confidence Bound, and Gradient Bandit. This way, training effort is decreased and it enables the use of the same set of versions with different platforms without sacrificing performance. We conclude that the proposed policies can identify the version to use without incurring substantial performance losses. Furthermore, when the user does not know enough details of the application to configure optimally the explore-then-commit policy usedy by other runtime systems, the more adaptable UCB policy can be used in its place.

Resumé

Les architectures informatiques modernes sont très complexes, nécessitant un grand effort de programmation pour obtenir toute la performance que le matériel est capable de fournir. En effet, alors que les développeurs connaissent les optimisations potentielles, la seule façon possible de dire laquelle est le plus rapide pour une plate-forme est de le tester. En outre, les nombreuses différences entre deux plates-formes informatiques, dans le nombre de cœurs, les tailles de cache, l'interconnexion, les fréquences de processeur et de mémoire, etc, rendent très difficile la bonne exécution du même code sur plusieurs systèmes. Pour extraire le plus de performances, il est souvent nécessaire d'affiner le code pour chaque système. Par conséquent, les développeurs adoptent l'autotuning pour atteindre un certain degré de performance portable. De cette façon, les optimisations potentielles peuvent être spécifiées une seule fois et, après avoir testé chaque possibilité sur une plate-forme, obtenir une version haute performance du code pour cette plate-forme particulière. Toutefois, cette technique nécessite de régler chaque application pour chaque plate-forme quelle cible. Non seulement cela prend du temps, mais l'autotuning et l'exécution réelle de l'application diffèrent. Des différences dans les données peuvent déclencher un comportement différent, ou il peut y avoir différentes interactions entre les fils dans l'autotuning et l'exécution réelle. Cela peut conduire à des décisions sous-optimales si l'autotuner choisit une version qui est optimale pour la formation, mais pas pour l'exécution réelle de l'application. Nous proposons l'utilisation d'autotuning pour sélectionner les versions du code pertinentes pour une gamme de plates-formes et, lors de l'exécution de l'application, le système de temps d'exécution identifie la meilleure version à utiliser à l'aide de l'une des trois politiques que nous proposons: Mean, Upper Confidence Bound et Gradient Bandit. De cette façon, l'effort de formation est diminué et il permet l'utilisation du même ensemble de versions avec différentes plates-formes sans sacrifier les performances. Nous concluons que les politiques proposées peuvent identifier la version à utiliser sans subir de pertes de performance substantielles. De plus, lorsque l'utilisateur ne connaît pas suffisamment de détails de l'application pour configurer de manière optimale la politique d'exploration puis de validation utilisée par d'autres systèmes de temps d'exécution, la politique UCB plus adaptable peut être utilisée à sa place.

During the period of elaboration of this work, the author received financial support from CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Contents

1 Introduction

Parallelism is pervasive in performance-critical applications. It is essential for a diverse array of scientific and industrial applications, such as geophysics, engineering, medicine, weather prediction, etc. Even everyday applications like a browser make use of parallelism to deliver the responsiveness users come to expect and that sequential programming often cannot. While initially restricted to High-Performance Computing (HPC), today parallelism permeates virtually all of computing – from supercomputers with millions of cores, with up to 10 million cores in the June 2019 Top500 [Don19] list of supercomputers, to personal computers and mobile phones.

Parallelism became almost omnipresent in modern computing due to the limits of what can be achieved sequentially in today's computer architectures. In the early days of computing, advances in manufacturing allowed, for instance, the trade of a small adder for a larger but faster adder. An increase in frequency allowed a CPU to perform faster without any major changes in its logic. However, there are limits to the performance gains these venues can provide [TW17], and the closer we get to these limits the lower are the benefits. Nowadays, large leaps in sequential performance are unlikely [Rah+16] without radical architectural changes, and further increases in frequency are difficult due to resulting in higher power requirements [SQP08], aggravating issues like heat dissipation [Ven+10]. As a consequence, the extra area in a die tends to be used to fit an increasing number of computer cores, improving performance despite the limited gains in sequential speed.

Yet, as computational power increases so do the computational requirements, with the use of more accurate models and the possibility to solve larger problems which were previously unfeasible. While parallel architectures help to provide much-needed increased computational power, they by no means make efficiency less of a concern.

With more CPU cores available, the pressure increases on resources which previously were not bottlenecks. For example, a large number of processing units can saturate the available memory bandwidth [LLS08; Hag+16]. One solution to this is to make use of Non-Uniform Memory Access (NUMA), effectively splitting the

memory into multiple regions, increasing total memory bandwidth and reducing latency. However, although necessary, these changes are not without consequences. To better exploit the characteristics of the platform, applications must take into account the characteristics of the computation, for instance by grouping accesses to nearby memory locations into cores which share a cache level, or storing data in the memory node closest to the cores who access it the most. All this further adds to the complexity of often already fairly complicated applications. Furthermore, the use of several computing units makes it effectively impossible for the developer of an application to manually manage all threads effectively. Several runtime systems exist for aiding the developer in managing the threads, such as OpenMP [Boa18], StarPU [Aug+11], OmpSs [Dur+11], Cilk [Blu+96], Intel TBB [Rei07], etc. The runtime abstracts many details of parallel computing, in much the same way a compiler abstracts the architectural details of the target CPU, enabling the developer to make use of the computational resources without having to specify exactly how. The ability of a runtime system to dynamically adapt the use of the resources during the computation is considered imperative to overcome the challenges of parallel computation [Bro+10; Luc+14; Acu+14].

1.1 Scientific Context

Modern computer architectures are highly complex. Although these architectures can provide high performance, an application must take advantage of architectural details before it can fully benefit from the architecture's capabilities.

Tuning an application for a specific hardware platform is a difficult undertaking. Code changes that improve one metric are often detrimental to another. For instance, improving Instruction Level Parallelism (ILP) through loop unrolling increases the size of the code, a tradeoff that can be advantageous or disadvantageous depending on both the application and the platform.

Autotuning can aid in deciding which of the several possible optimizations are appropriate for a specific platform. However, an application still must be tuned for each of the platforms before it can take advantage of the optimizations. Applications that depend heavily on libraries for their computation can benefit from autotuning with libraries like ATLAS [WPD01], SPIRAL [Xio+01], FFTW [FJ98], and PhiPAC [Bil97], which then requires tuning not the application but every relevant library for each platform. While not as fine-grained as the approach from the libraries, runtimes like OmpSs [Dur+11] and StarPU [Aug+11] can dynamically tune a computation by offloading more, or less, work to an accelerator based on performance and system load.

1.2 Objectives

The main objective of this work is to facilitate the use of highly tuned implementations of the performance-critical parts of task-parallel OpenMP applications. Tuning an application is challenging not only due to the vast complexities of modern computing platforms, with varying sizes of cache, different memory access speeds depending on data locality, threads influencing one another, etc, but also because all these complex interactions vary between platforms. Consequently, the performance tuning for one system does not guarantee performance on a different system. Usually, to obtain performance on a new system one must tune the application specifically for that system. We aim to allow an OpenMP application to identify, while it executes, which particular tunings are beneficial, while at the same time avoiding those unsuitable for that specific platform.

1.3 Contributions

The first contribution of this thesis is the introduction of multi-versioned tasks to OpenMP. We propose three task selection policies, one based on averages, like some of the policies of other runtime systems, and the two others based on statistically-sound methods for multi-armed bandit problems, which as far as we are aware are not employed by any task-parallel runtime. Lastly, we integrate our proposal in LLVM, more specifically in the LLVM OpenMP runtime and the Clang compiler. These additions to LLVM can, in turn, be used to improve the performance of OpenMP applications either with manually-tuned kernel versions or with kernel versions obtained through autotuning.

1.4 Outline

Chapter 2 presents the scientific context of this thesis. That chapter describes multicore architectures, with their main advantages and caveats. It also shows the parallel programming paradigm, with some details on runtime systems that aid the application in making full use of the resources available. The chapter also details performance analysis and some statistical methods that can be employed. The chapter also explains machine learning, and some of the difficulties therein. Another topic approached by that chapter is that of autotuning, which consists of programmatically tuning some software for some hardware platform. Lastly, this chapter expounds on the multi-armed bandit problem and its relevance to this work.

In sequence, Chapter 3 presents our contributions. The chapter details the scientific problem and its relevance. It also presents our extensions to the LLVM compiler and changes made to the scheduler of the OpenMP runtime.

Chapter 4 presents the experimental evaluation of our proposal. It describes the experimental methodology employed, with details of the platforms used. It also presents the benchmarks we use, with their relevant parameters, and the different scenarios used with these benchmarks. The chapter also shows our experimental results for a single thread, multi-thread, and multi-thread with multiple sockets.

Chapter 5 presents related work. It details the policies employed by runtime systems that handle multi-versioned tasks.

Lastly, Chapter 6 presents our conclusions.

2 Background

In this chapter, we review basic concepts required for understanding this thesis. Section 2.1 briefly describes multicore architectures. Section 2.2 presents an overview of the parallel programming paradigm and a few parallel runtimes. Section 2.3 presents statistics concepts which are employed in subsequent chapters. Section 2.4 presents an overview of machine learning. Section 2.5 briefly explains the basic concepts of autotuning. Section 2.6 explains the multi-armed bandit problem. Finally, Section 2.7 concludes this chapter with our remarks.

2.1 Multicore Architectures

Performance gains in microprocessors due to higher clock speed have essentially stagnated for several years now due to the diminishing returns provided by further increases in clock frequencies. This breaks a historical trend where new microprocessor models almost always featured a higher clock frequency than their predecessors. The main reason for this change is the impact of the frequency on power consumption. A microprocessor's power consumption can be split into three components: dynamic, static and short-circuit power [Zhu+13]. Dynamic power dominates the power dissipation, being followed by static power and then short-circuit power. The clock frequency is given by

$$f = k \cdot \frac{(V_{dd} - V_{th})^2}{V_{dd}}$$

for some constant k, a supply voltage V_{dd} and a threshold voltage V_{th}, but while the frequency is only linearly related to the supply voltage the dynamic power also depends on frequency. The dynamic power dissipation is given by

$$P_d = C_l \cdot N_{sw} \cdot V_{dd}^2 \cdot f$$

where C_l is the load capacitance and N_{sw} the average number of circuit switches per cycle. Due to the dependency of the clock frequency on the supply voltage, power consumption is cubically related to the clock frequency.

As higher frequencies come with the price of higher power consumption [MM17], increased running costs and requiring improved power dissipation, power is a

major obstacle in the way of ever-increasing clock speeds. As Figure 2.1 shows with the green squares, clock speeds have stagnated and single-thread performance has seen more modest improvements in the last few years. In the same figure, the red dots show the number of transistors continues increasing. While many years ago the extra space was filled with larger but faster logic, besides bigger caches, now much of it goes to increasing the number of cores. However we are approaching physical limits and Moore's Law [Moo+65] is beginning to fault, as evidenced by Intel turning away from their tick-tock strategy and rising transistor costs [KHF18], which could make further improvements difficult. Despite this, performance gains have not stopped and the number of cores continues to steadily increase, a trend that should continue at least while Moore's Law holds.

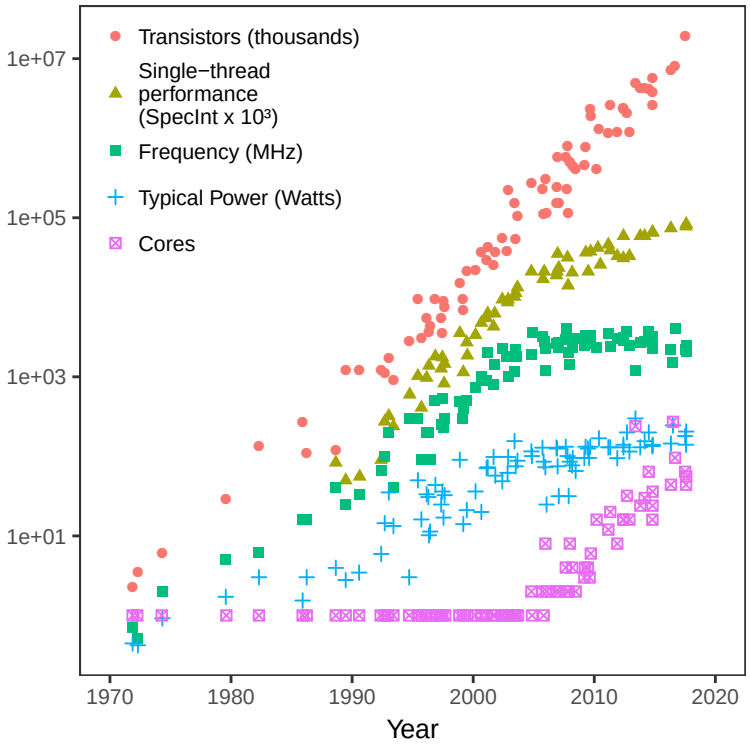

Figure 2.1.: Number of transistors, single-thread performance, clock frequency, power consumption and number of cores for microprocessors through the years. Data sources: [Hor+13; Rup18].

This increase in the number of cores allows for still large performance gains with parallel computing despite the deceleration of single-thread performance improvements. Although multicore has made parallelism more popular relatively recently, parallelism is an approach that has been employed several times before. Instruction-level parallelism (ILP) runs instructions whose operands are independent of each other in parallel; vectorial instructions allow applying the same operand to multiple data (Single Instruction Multiple Data, SIMD [Fly66]); and even in the early days of computing tabulators and multipliers were used in parallel, in a kind of

manually-operated pipeline, to speed up the computation and make better use of the very expensive resources.

ILP improves performance transparently to the programmer as the hardware does all the required work of verifying dependencies and availability of computational resources. While not as transparent, vectorial instructions can improve performance semi-transparently to the programmer as these are often used by libraries and, in some cases, automatically by the compiler. Multicore, however, is anything but transparent. In order to make use of more than a single core, the programmer has to modify the software to that end. Cores must compete for shared resources like the shared L3 cache between cores and memory, especially in machines with a Non-Uniform Memory Architecture (NUMA). As the cores can affect one another this makes the effective use of the available computational power an even harder to reach goal.

Much of the area in a modern microprocessor is occupied by cache. A level 1 (L1) cache attempts to minimize latency, which means it must be located near the units which access the cache. That, in turn, imposes constraints on the chip's design, severely limiting how large a L1 cache can be while providing a low latency. Due to the low latency provided by the L1 cache, it tends to be private to the core. On the other end, a last level cache (LLC), L3 in Intel or AMD processors, has far more relaxed constraints, allowing the L3 cache to be much larger than the first level. Unlike L1, L3 tends to be shared among the cores. A private cache is better suited for workloads which present very little data sharing [Zha+08] as it prevents the cores from invalidating data that is still in use by other cores. A shared cache is preferable when cores access the same data, as it spares the cores from having to make a much slower memory access and reduces data replication.

While the growing number of cores in a single package undoubtedly helps increase computational power it imposes further pressure on memory bandwidth. Which in turn worsens the memory wall problem [WM95], which is caused by the different rates at which microprocessor and DRAM speeds increase. As microprocessors improve faster than memory does, the relative time the processor spends waiting for memory keeps increasing. One way to reduce the pressure on memory bandwidth is to have the memory distributed over the system instead of centralized. This way, each memory bank can be close to one socket, granting that processor faster memory access. This design is named Non-Uniform Memory Architecture (NUMA) and is illustrated in Figure 2.2. While it is shared memory, as any socket can access any other socket's memory, accessing another socket's memory has a higher latency than local accesses. This happens because remote accesses must pass through an interconnection network to reach their destination, such as FSB, HyperTransport [CH07], and QuickPath Interconnect (QPI) [Zia+10]. In the figure,

the interconnect linking two sockets is represented by an edge. Furthermore, access costs to different nodes may differ, that is, it may be faster for a node A to access node B's memory than for it to access C's.

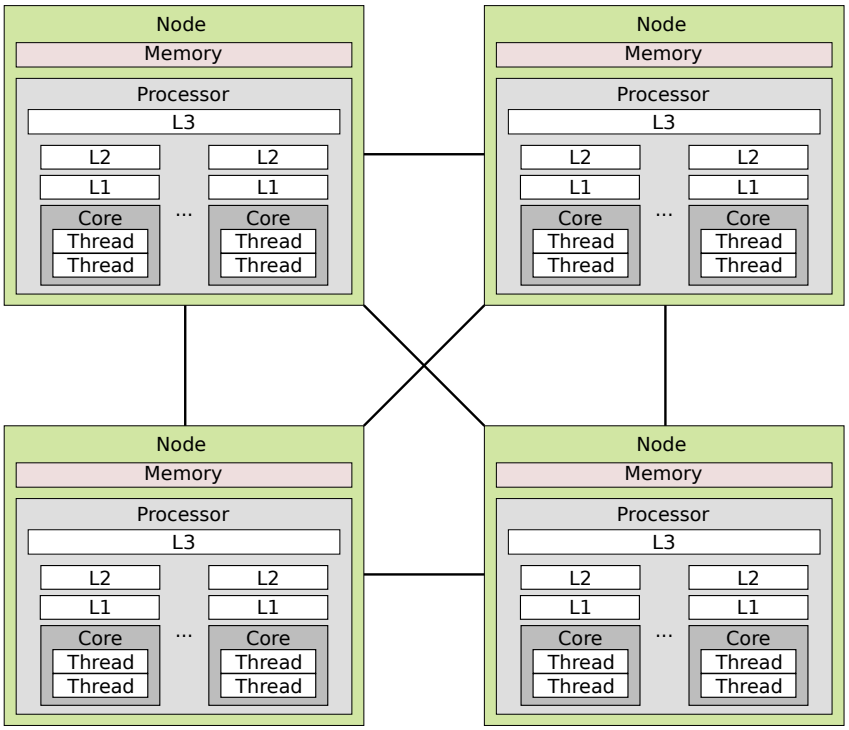

Figure 2.2.: Non-Uniform Memory Architecture. While any sockets can access the memory of a remote node the latency to do so is not the same for all remote nodes.

Using more than a single core adds several complexities to an application that do not exist in a sequential environment. For example, let's say some memory location contains a counter, which has its value read, incremented by one, and written back every time some event happens. If two threads attempt to update this counter at the same time, the two threads may read the value n concurrently and write back $n+1$ instead of the expected $n+2$. In order to prevent this type of problem it is necessary to have some form of synchronization. Parallel applications also must divide the work among the threads, and, in order to remain efficient, account for memory location when splitting the work such that threads are assigned data that is close to them, and, thus, quicker to access. Since the use of multithreading requires several changes in the way an application is developed, different solutions have been proposed to deal with these difficulties, some of which are described next.

2.2 Parallel Programming

This section describes the parallel programming paradigm, giving an overview of some parallel programming models and implementations. Over the years several techniques have been developed to handle parallel programming.

Implicit parallelism allows the application to make use of several cores or nodes without requiring the programmers to explicitly control their use. It can be present in many functional programming languages since the lack of side-effects allows functions which do not depend on each other to run in parallel. Implicit parallelism reduces the effort required by the programmer to make an application run in parallel. However, it often results in either too few or too many [Hon] tasks being created, resulting, respectively, in idle resources or a high overhead when compared to more explicit methods.

The message passing model allows different nodes to communicate by sending and receiving messages to one another. It is used for instance to communicate between nodes in a cluster. One of the most well-known standards that use message passing is the Message Passing Interface (MPI), and it can be used with several programming languages, like C, C++, Fortran, R, Java and Python.

In shared memory environments, where every computing unit has access to all memory, such as between cores within a single machine, the fork-join model is often preferred. With this model, a thread forks execution of some subproblem and continues executing. As it is a model that targets shared memory environments data does not need to be directly transferred between threads. However in environments with Non-Uniform Memory Access (NUMA) for instance this direct access to memory can lead to inefficient memory use, since if care is not taken threads may spend a considerable amount of time accessing non-local data.

There are many implementations of the fork-join model. Some examples are:

2.2.1 POSIX Threads

The Portable Operating System Interface (POSIX) is a family of standards created by IEEE, ANSI and ISO [Gal95]. POSIX was created with the aim of providing certain compatibility between different operating systems through the use of the same set of APIs. While complete portability free from OS-specific code is still often not possible, efforts such as POSIX reduce the cost of porting an application between two POSIX-compliant operating systems. Several operating systems are at least

partly POSIX-compatible. For example AIX, Android, FreeBSD, Haiku, INTEGRITY, Linux, macOS, NetBSD, OpenBSD, OpenSolaris, Solaris, UP-UX and Windows.

POSIX Threads, or pthreads, is the POSIX API for multithread programming. It specifies thread creation and synchronization through mutexes or condition variables. It is often used along with POSIX semaphores, which are not part of the same standard. POSIX Threads is an API meant for system programming, being used by several other libraries. It is ill-suited for programming an application in as it is a low-level API, making it difficult to write performance-portable code.

Listing 1 shows an example of a simple POSIX Threads program. In that code two POSIX features are used. pthread_create in lines 10-11 creates a new thread that executes the function fibonacci and then quits. That function also receives data through a pointer to n (n+1, in line 11), and writes the thread identifier in the variable tid[0] (tid[1] for line 11). The other feature used is pthread_join, which waits for the thread with the given thread identifier to finish executing before returning and in this case is used to ensure the data in n[0] and n[1] read afterwards has been written.

```
1  void * fibonacci(void *a){
2    pthread_t tid[2];
3    int n[2];
4    n[0] = *(int*)a;
5    if(n[0] < 2){
6      return 0;
7    }
8    n[0] = n[0] - 1;
9    n[1] = n[0] - 1;
10   pthread_create(tid+0,NULL,fibonacci,n+0);
11   pthread_create(tid+1,NULL,fibonacci,n+1);
12   pthread_join(tid[0],NULL);
13   pthread_join(tid[1],NULL);
14   *(int*)a = n[0] + n[1];
15   return 0;
16 }
```

Listing 1: Example of POSIX Threads code for Fibonacci.

2.2.2 OpenMP

Open Multi-Processing [Boa18] (OpenMP) is a shared-memory parallel programming standard supported by several compilers in different operating systems and languages. Initially, OpenMP only supported parallel loops. Parallelizing a loop where all iterations are independent and can be run in any order can greatly speed up the computation. Different mappings of loop iterations to threads are available through the use of different scheduling policies. The “static” scheduler, for example,

does this mapping statically, dividing the iterations evenly (when possible) among the threads. This behaviour is helpful for data locality in NUMA systems using a first-touch NUMA policy. This policy delays memory allocation to the first time a write is made to it, and, then, allocates the memory in the node of the thread which is attempting to write. The “static” scheduler can help locality since loops with the same size will have their iterations mapped to the same threads, which in a simple scenario means the threads always work on the same data, which is local because of the first-couch policy. However, when iterations have different workloads a static mapping will result in idle threads while other threads still have enqueued work to do. The “dynamic” scheduler handles this scenario by dynamically distributing the iterations to the threads during execution, such that a thread who finished all the iterations assigned to it grabs and starts executing the next chunk of iterations that have not been assigned to any thread yet. This however adds the overhead of threads having to grab perhaps several new chunks of iterations. To reduce the overhead one can increase the chunk size, so that threads will grab a larger number of iterations each time. However a large chunk size may result in slowing down the computation as a few threads still have enqueued work to do while other threads are idle. To mitigate this issue OpenMP provides the “guided” scheduler. This scheduler is similar to the "guided" scheduler but it starts with large chunks and slowly reduces their sizes, aiming to reduce the overhead while not causing too much work inbalance.

Parallel loops are useful for many applications, particularly when the same, or similar, operations have to be applied to multiple data. However there are severe limitations to this model. The OpenMP parallel loop construct can easily lead to idle resources as threads which have finished their computation must wait for other threads to finish, which is an larger problem when the data cannot be evenly distributed among the threads. Furthermore, in many cases it is necessary to run different operations in parallel, which can be difficult to do efficiently using a loop construct. The 3.0 version of the OpenMP standard introduced support for task programming. The 4.0 version of the OpenMP standard added support for data dependencies in tasks. Tasks tend to be heavier than parallel loops, however tasks can better handle more dynamic workloads as they reduce the number of synchronization points necessary. The use of parallel loops or tasks without dependencies may result in idle computational resources, but with the use of dependencies a task may start executing as soon as its data is available and an idle thread is available. Due to the several advantages of tasks there is far more research effort today in improving the OpenMP tasking model than on improving its parallel loops.

Listing 2 shows an example of OpenMP code for a parallel Cholesky decomposition, also known as Cholesky factorization, which is used for instance by Monte Carlo methods used to simulate fluids, simulate cellular structures, calculate business

risks, calculate multidimensional definite integrals with complex boundaries, etc. A Cholesky decomposition has the form $A = LL^*$ and consists of factorizing the Hermitian, positive definite matrix A into the product of two matrices, the lower triangular matrix L and its conjugate transpose L^*. Line 2 specifies the next block should be run in parallel by a team of n threads for some previously specified n. Line 3 indicates only a single thread should execute this block. Lines 5, 8, 14 and 19 specify the next block is run as a task. In that last case, the "depend" clause is used to specify dependencies. Tasks with an "out" and "inout" clause fulfil the dependencies of tasks created later and which have the same variables in their "in" and "inout" clauses. Tasks with "in" and "inout" clauses must wait for these dependencies to be fulfiled before starting execution.

```
1  void cholesky(int ts, int nt, double *A[nt][nt]){
2  #pragma omp parallel
3  #pragma omp single
4    for(int k = 0; k < nt; ++k){
5  #pragma omp task firstprivate(k) depend(inout: Ah[k][k])
6      potrf(Ah[k][k], ts, ts);
7      for(int i = k + 1; i < nt; ++){
8  #pragma omp task firstprivate(i,k) depend(in: Ah[k][k]) \
9                                     depend(inout: A[k][i])
10       trsm(Ah[k][k], Ah[k][i], ts, ts);
11     }
12     for(int i = k + 1; i < nt; ++i){
13       for(int j = k + 1; j < i; ++j){
14 #pragma omp task firstprivate(i,j,k) depend(in: Ah[k][j], \
15                                                Ah[k][i]) \
16                                       depend(inout: Ah[j][i])
17         gemm(Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);
18       }
19 #pragma omp task firstprivate(i,k) depend(in: Ah[i][i]) \
20                                     depend(inout: Ah[k][i])
21       syrk(Ah[k][i], Ah[i][i], ts, ts);
22     }
23   }
24 }
```

Listing 2: Example of OpenMP code for Cholesky decomposition.

OpenMP requires compiler support to convert its "pragma" constructs into the appropriate code, which creates threads, copies or gathers data, waits for other threads to finish, schedules new tasks for execution, etc. Much of this code consists of function calls to the OpenMP runtime, which is the library responsible for providing all the necessary support for OpenMP applications during their execution.

libGOMP [Lib15] is the OpenMP runtime used by GCC. It keeps track of dependencies by means of a hash table mapping the data to the last writer to that data [Lim+17]. Threads in the same parallel region push tasks to a shared dequeue, requiring synchronization and in turn creating a bottleneck. In order to keep task

execution order similar to the sequential execution tasks are inserted after their parent. To reduce the task creation overhead, OpenMP uses task throttling to serialize task creation when there is too large a number of pending tasks relative to the number of threads.

The libOMP runtime was initially developed by Intel for its compilers. Initially proprietary, the runtime has since been made open source and was adopted by the LLVM project for use with the Clang compiler. GCC's codebase is complex in great part due to its legacy code. LLVM, however, is a much newer project, enabling it to more easily make use of advancements made in compilers since and resulting in a much more accessible codebase, making it easier to extend LLVM when compared to GCC. Like libGOMP, libOMP keeps track of task dependencies with a hash table. The task scheduler is based on Cilk's work-stealing algorithm but uses locks for the serialization of dequeue operations. Different from libGOMP, the task dequeue is distributed among the threads. Memory allocated for task creation uses a fast thread memory allocator. The task throttling algorithm uses a bounded size dequeue to handle a large number of tasks being created.

2.2.3 OmpSs

OmpSs [Dur+11] continues the work of StarSs, extending SMPSs and GPUs and integrating the heterogeneity support into a single programming model. Like StarSs, OmpSS makes use of the Mercurium compiler and the Nanos++ runtime. Many of the features first introduced in OmpSs are later incorporated in OpenMP.

The OmpSs runtime provides a few different scheduler policies. The *breadth first* policy uses a single queue shared by all threads, ordered by default as a FIFO. The *distributed breadth-first* policy is similar to *breadth-first* but it uses a queue per thread. When this local task is empty the thread attempts to execute the current task's parent, if the current task is from a different thread's queue). If that task cannot be executed the thread attempts to steal a ready task. The *work first* policy has one ready queue for each thread. When a new task is created it is executed immediately, and the parent task is placed into the current thread's ready queue. With the default parameters, this policy is equivalent to the Cilk scheduler. The *socket-aware scheduler* policy assigns top-level tasks to a user-defined NUMA node. Nested tasks, in turn, are assigned to the NUMA node of their parent. The policy supports but does not requires work-stealing. With work-stealing a thread may only steal from neighbouring nodes, or, alternatively, steal from a random node or steal top-level tasks. The *bottom level-aware scheduler* policy was made for single-ISA heterogeneous machines with two types of cores, like ARM's big.LITTLE which features fast cores that consume more power and slower cores which require less

power. It uses two queues, one for each kind of core, sorted by task priority. With this policy, fast cores are assigned tasks which are performance-critical in the task dependency graph. By default work-stealing is enabled only for fast cores.

The *versioning* policy supports multiple versions of a single task. The profiling of the multiple versions is done automatically. The policy assigns a task to its earliest executor, but when the number of tasks is large enough idle threads can execute them even if they are not the earliest executors.

Listing 3 shows an example of OmpSs for computing Cholesky decomposition. Lines 1, 3, 5 and 7 specify the function below each of them, when called, spawns a task. In the pragma used by those lines the variables in the clause "in" are input dependencies. Likewise, variables inside an "inout" clause specify both a dependency of that variable in a previously created task and that it satisfies that dependency of a future task. A task only begins executing when all its dependencies ("in") have been satisfied by tasks created previously (with "inout" or "out"). Lastly, line 20 specifies the worker will wait for the tasks it created previously to finish before continuing. The use of pragmas allows the program to still compile and execute, sequentially, when compiled by compilers that do not support the pragmas.

```
1  #pragma omp task inout([ts][ts]A)
2  void omp_potrf(double * const A, int ts, int ld){}
3  #pragma omp task in([ts][ts]A) inout([ts][ts]B)
4  void omp_trsm(double *A, double *B, int ts, int ld){}
5  #pragma omp task in([ts][ts]A) inout([ts][ts]B)
6  void omp_syrk(double *A, double *B, int ts, int ld){}
7  #pragma omp task in([ts][ts]A, [ts][ts]B) inout([ts][ts]C)
8  void omp_gemm(double *A, double *B, double *C, int ts, int ld){}
9  void cholesky(int ts, int nt, double *A[nt][nt]){
10   for(int k = 0; k < nt; ++k){
11     omp_potrf(Ah[k][k], ts, ts);
12     for(int i = k + 1; i < nt; ++)
13       omp_trsm(Ah[k][k], Ah[k][i], ts, ts);
14     for(int i = k + 1; i < nt; ++i){
15       for(int j = k + 1; j < i; ++j)
16         omp_gemm(Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);
17       omp_syrk(Ah[k][i], Ah[i][i], ts, ts);
18     }
19   }
20 #pragma omp taskwait
21 }
```

Listing 3: Example of OmpSs code for Cholesky decomposition. Source: OmpSs Documentation [Cen18].

2.2.4 StarPU

StarPU [Aug+11] is a task programming runtime initially made for heterogeneous architectures. It allows the user to define a set of kernel versions for CPUs and accelerators. When a task is launched and its dependencies met the runtime selects which of the provided versions to run. Data transfers between the main memory and accelerators are done transparently to the programmer. With the KSTAR [Aum+18], or Klang-OMP, source-to-source OpenMP compiler OpenMP source code can transparently use the StarPU library instead.

In order to optimize the execution in different scenarios, StarPU provides a large set of schedulers. The schedulers are split into two groups, those with and those without a performance model. The performance model provides an estimation of how long a task will run for, without having to execute it first.

StarPU allows the application itself to provide a performance model for each kernel. It is also possible to use a model which measures the time taken by each execution of each kernel and no further information, which works when the workload of each kernel does not change during the execution. If the workload of each kernel is not constant through the execution a regression with the form $a \cdot n^b$ or $a \cdot n^b + c$ can be used instead. Due to the cost of computing non-linear regressions, they are only done at the end of the execution — so during the execution only the recorded performance history is used by the scheduler and new measurements will only be used by future executions.

The group of schedulers that do not require a performance model consists of five schedulers: eager, prio, random, work-stealing and locality work-stealing. The eager and the prio schedulers both have a single global queue shared by all workers. The two differ in that the eager policy puts any task with a non-zero priority at the front of the queue and the prio policy sorts the tasks according to their priority. The other three schedulers have one queue per worker. The random scheduler distributes the tasks randomly among the workers. While faster workers will receive more tasks this policy ignores how well-suited a task is for a worker. As such if a worker is faster than another but only with one type of task it will still receive tasks it cannot compute as quickly. With the work stealing scheduler a worker executes first the tasks it created. Once a worker's queue is empty, the worker attempts to steal tasks from a random worker. The locality work-stealing scheduler aims to improve the locality of the work stealing scheduler. Idle workers steal not from a random worker but from a neighbouring worker, accounting for priorities.

Listing 4 shows an example of vector scaling in StarPU. Lines 1 and 2 provide two versions of the kernel. Lines 3-9 group these two functions into a codelet, which can be called afterwards and will use one of the provided versions. Line 8 defines the data used by the task is both read and written to. Line 15 initializes StarPU with the default parameters. Line 16 registers the data with the StarPU runtime, enabling StarPU to transparently manage data movements between host and devices. Line 22 specifies the codelet the task will use. Lines 23-25 define the data that will be passed to the task. Line 26 submits the task, and, since it is a synchronous task, waits for it to finish execution. Line 27 tells StarPU the location of the data does not need to be tracked anymore. Finally, line 28 shuts down StarPU.

```
1  void scal_cpu_func(void *buffers[], void *_args) { ... };
2  void scal_sse_func(void *buffers[], void *_args) { ... };
3  static struct starpu_codelet cl = {
4    .where = STARPU_CPU;
5    .cpu_funcs = { scal_cpu_func, scal_sse_func },
6    .cpu_funcs_name = { "scal_cpu_func", "scal_sse_func" },
7    .nbuffers = 1,
8    .modes = { STARPU_RW }
9  };
10 void vector_scaling() {
11   float vector[NX];
12   unsigned i;
13   for (i = 0; i < NX; i++)
14     vector[i] = 1.0f;
15   starpu_init(NULL);
16   starpu_data_handle_t vector_handle;
17   starpu_vector_data_register(&vector_handle, STARPU_MAIN_RAM,
18     (uintptr_t)vector, NX, sizeof(vector[0]));
19   float factor = 3.14;
20   struct starpu_task *task = starpu_task_create();
21   task->synchronous = 1;
22   task->cl = &cl;
23   task->handles[0] = vector_handle;
24   task->cl_arg = &factor;
25   task->cl_arg_size = sizeof(factor);
26   starpu_task_submit(task);
27   starpu_data_unregister(vector_handle);
28   starpu_shutdown();
29 }
```

Listing 4: Example of StarPU code for vector scaling. Simplified from the example in the StarPU Handbook [BCI19].

While all the mentioned APIs have their pros and cons, one important attribute is how each of them impacts performance. Although performance is a more objective attribute to evaluate than, for instance, ease of use, its evaluation demands some attention in order to avoid reaching invalid conclusions that do not hold in the general case.

2.3 Performance Analysis

This chapter describes the statistical background necessary for the analysis of our experimental results and some of our approaches. At first we motivate the reasons for this requirement and, afterwards, we explain the statistical methods employed.

The analysis of HPC experiments is often limited to a direct comparison of the means of the points being compared. Although simple this approach can be misleading in some cases. The mean is not a robust measurement — it can be strongly affected by even a single point which is far from most other points. Furthermore, by limiting ourselves to the mean it becomes impossible to know how far most results are from it, and, consequently, one cannot tell how well the mean represents the data.

To illustrate this point, Figure 2.3 shows two ways to implement some hypothetical computation, A and B. Both versions have the same mean execution time, represented by the red dot. However, by looking at each of the 100 executions, shown as the black dots, we can see in this case the two samples are actually very different from each other: the executions of A are clustered close together, while those of B are more spread. That is, B presents a higher variability than A. As the two means are the same, A and B have the same expected value ($\mathbb{E}$). However the two only behave the same in the long term. If we were to take a single measurement, B shows itself much more rewarding and risky than A since it has a much higher chance of measurements farther from the mean.

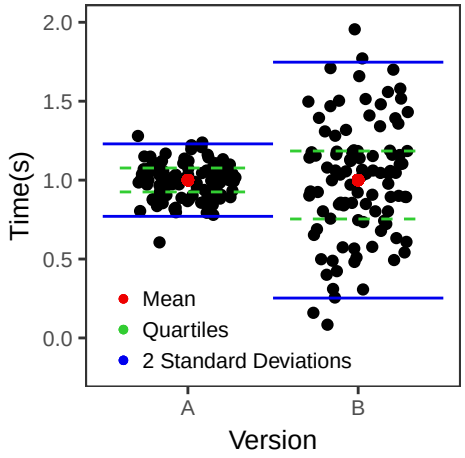

Figure 2.3.: Example of how the mean can mask details such as the shape of the measurements.

Figure 2.4 depicts a scenario where a computing kernel *K0* is being executed by thread 0 while thread 1 is executing the kernel *K1*, with the horizontal axis showing at which point in time each kernel began and finished executing. Once

thread 1 finishes executing *K1* it can begin executing another kernel, *K2*. The *K2* kernel, however, can be computed in two different ways, *K2(A)* and *K2(B)*. These are shown, respectively, by the purple and orange bars, which end at the average time each should finish executing. These averages come from Figure 2.3 and are, again, the same. Another kernel, *K3*, is in the critical path of the application. That is, *K3* directly affects the execution time of the application and any delay in its execution will negatively impact performance as it will increase the makespan. In that case, if *K3* depends on *K0* and *K2*, the choice between *K2(A)* and *K2(B)* can impact execution time despite both sharing the same mean. The red lines show when 95% of executions of *K2* should finish with each version. As *K2(B)* has much higher variability than *K2(A)*, *K2(B)* has a higher chance of finishing not only after *K2(A)* but after *K0* as well. For the same reason *K2(B)* also has a higher chance of finishing earlier than *K2(A)*. However in this particular case *K2* finishing earlier is not advantageous from a performance perspective as *K3* still has to wait for *K0* to begin executing, making *K2(A)* better than *K2(B)* despite both having the same average execution time. There are several statistical methods to measure the variability of a sample, some of which will be briefly discussed in the next paragraphs. These other measures add more information but do not replace the mean as it still provides valuable information.

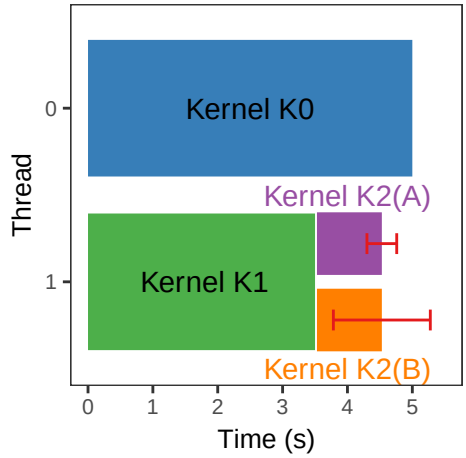

Figure 2.4.: Example showing a parallel exectution. Thread 0 executes kernel K0 in parallel with kernels K1 and K2 in thread 1. K2 has two versions available, K2(A) and K2(B), shown by the purple and orange bars respectively. In K2(A) and K2(B) the end of the bar indicates the mean execution time of that kernel based on past executions, depicted in Figure 2.3. The red lines represent two standard deviations and show where 95% of the executions should fall.

The *range* tells how much difference can be found among points in the sample. It is given by the difference between the maximal and minimal measurements taken ($max(x_i) - min(x_i)$). Like the mean, the range is a very sensitive measure as it only takes two points, and at the extremes which are where few points lie in the most common distributions. For example in the Figure 2.3 the ranges of A and B are,

respectively, 0.7 and 1.9. One can also limit the range to a portion of the sample instead, making the measurement less sensitive to extreme and improbable values. If we split the data points into quartiles, that is, if we group together the 25% lowest values from the sample, then the next 25% of points and so on, the interquartile range (IQR) gives a range. Instead of calculating that range from the two extreme values of the sample the IQR takes the largest and smallest values inside the two middle quartiles. In other words, the IQR takes the extreme values but only from the 50% of points closest to the middle of the sample — 25% from among the points immediately before the median and the other 25% from the points directly above it. This range is shown in Figure 2.3 by the dotted green lines.

The *interquartile range* is a more robust measure of variability than the sample range as the IQR is not as sensitive to a few extreme values, however it implies in ignoring a large number of points. These points that are far from the average, the outliers, are often discarded. There is no general rule on what defines an outlier, although there are some methods which are often employed. Tukey's inner fences [Tuk77] define an outlier as all points below $Q_1 - 1.5IQR$ or above $Q3 + 1.5IQR$. Dixon's Q-test [DD51] can be used to identify a single outlier in each tail of a Gaussian-distributed sample. It is a simple test, where values in both extremes are compared with their neighbors and the range. The N values in the sample are ordered such that $x_1 < x_2 < \cdots < x_n$, then the Q-value is calculated at the two tails:

$$Q = \frac{x_N - x_{N-1}}{x_N - x_1}$$

for the largest value and

$$Q = \frac{x_2 - x_1}{x_N - x1}$$

for the smallest value. The Q-value is then compared with a critical value for the desired confidence level which will decide whether the extreme values should be considered outliers, and possibly discarded. The Grub test [Gru69; Bar+11] can detect a single outlier in the upper and the lower ends of a Gaussian-distributed sample. The N values in the sample are ordered such that $x_1 = min(x)$ and $x_n = max(x)$. Then two values are calculated:

$$G_l = \frac{\overline{x} - x_1}{s}$$

where s is the sample's standard deviation, which is explained in the next paragraphs, for testing outliers at the lower tail and

$$G_u = \frac{x_n - \overline{x}}{s}$$

for the upper tail. If G_l or G_u is greater than

$$\frac{n-1}{\sqrt{n}}\sqrt{\frac{t^2_{\frac{\alpha}{2n},n-2}}{n-2+t^2_{\frac{\alpha}{2n},n-2}}}$$

where $t_{\frac{\alpha}{2n},n-2}$ is the critical value of the t-distribution with $n-2$ degrees of freedom and a significance level of $\frac{\alpha}{2n}$, the respective point is considered an outlier. Peirce's criterion [Pei52] can detect k outliers. For each value suspected of being an outlier $\frac{|x_i-\overline{x}|}{s}$ is calculated. If this value is greater than a critical value R, which depends on the number of measurements and the number of suspect points, the point is considered an outlier.

While rejecting outliers protects measurements like the mean from possibly undesired interference. There may be many sources of interference that the researcher may not want to look into. The equipment used for taking the measurements may not always be accurate, for instance, due to a dirty lens in the case of a telescope. There may be environmental factors that affect the measurements like radio noise. Or, in the case of a server, the temperature of the room it is located at, which in turn may prevent it from reaching higher clock frequencies. While some of these measurements may be very inaccurate, or even wrong, and do not represent the real phenomenon being measured, like in the case of the dirty lens, in other cases these may be normal — albeit unlikely — measurements. In those cases removing the measurements should be avoided. The United States' National Bureau of Standards for example states [FPP07] "Rejection of data on the basis of arbitrary performance limits severely distorts the estimate of real process variability. [...] Realistic performance parameters require the acceptance of all data that cannot be rejected for cause." Fortunately, a more robust statistical analysis can handle a certain number of outliers without altering the data. Some of the statistical tools used for that end are described next.

The *standard deviation* is a measure of how spread the data is around the mean. The larger the standard deviation, the more and farther the measurements are from the mean. If the data follows a Gaussian distribution like shown in Figure 2.5, 68% of all measurements will be within one standard deviation from the mean, 95% of all measurements will be within two standard deviations and 99.7% of all measurements will be within three standard deviations. In other words, in that scenario, 99.7% of all measurements are in the interval $[\overline{x}-3\cdot\sigma, \overline{x}+3\cdot\sigma]$, where σ is the standard deviation. Given a population of size N composed of the points $x_i, \forall i \in \{1,\ldots,N\}$ and with an average of

$$\mu = \frac{1}{N}\cdot\sum_{i=1}^{N} x_i$$

the standard deviation of that population is given by [FPP07]:

$$\sigma = \sqrt{\frac{1}{N} \cdot \sum_{i=1}^{N} (x_i - \mu)^2}$$

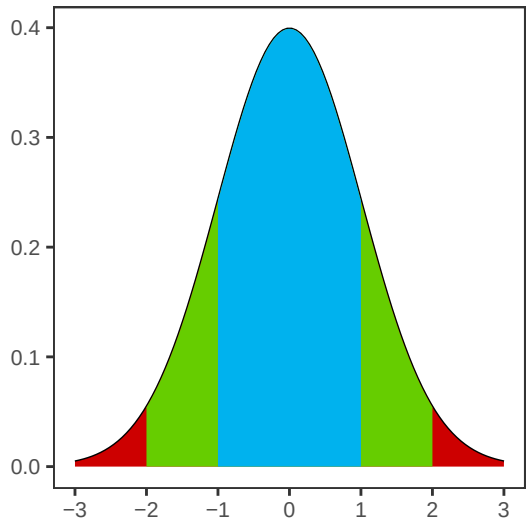

Figure 2.5.: Example of a Gaussian distribution. The area in blue shows the values under one standard deviation of distance from the mean, both above and below, which is where 68% of the values lie. Adding those to the area in green are the 95% of values within two standard deviations from the mean. Finally, if we add the area in red we have the 99.7% of values within three standard deviations from the mean.

In the more common case of calculating the standard deviation of a sample instead of the whole population, this is done slightly different. As the average of the sample, given by

$$\overline{X} = \frac{1}{N} \sum_{i=1}^{N} X_i$$

is not equal to the average of the population the root mean square deviation from the average would add bias to the standard deviation. This bias has to be compensated and the sample standard deviation is given by

$$s = \sqrt{\frac{1}{N-1} \cdot \sum_{i=1}^{N} (X_i - \overline{X})^2}$$

The *standard error* is similar to the standard deviation. However, instead of measuring the dispersion of a set of points, the standard error measures the variation around some statistic. For instance if we sample k values from some Gaussian

distribution we can expect the sum of these values to be, on average, $k \cdot \overline{x}$. Since sampling is a stochastic process sometimes this sum will be above, or below, the previously mentioned value. The standard error tells how likely are we to see values far from that expected value. The standard error of the mean (SEM) is given by

$$\sigma_{\overline{x}} = \frac{\sigma}{\sqrt{n}}$$

As the standard deviation of the population is often unknown this value can be estimated using the standard deviation of the sample with $\sigma_{\overline{x}} \approx s_{\overline{x}} = \frac{s}{\sqrt{n}}$. Notice this estimation of the population's standard error is biased and underestimates the population's standard error.

A *confidence interval* (CI) defines an interval in which some parameter of the population is found, with some confidence level. For example, if a sample from a gaussian distribution has a confidence interval of 100 ± 10 with a confidence level of 95% for the population mean, there is a 95% chance the real population mean is between 90 and 110. In this same example the 10, at a 95% confidence level, corresponds to the standard error times two, that is to say in that case the standard error is 5. With a confidence level of 95%, there is still a 5% chance the real mean is outside that interval. If instead we wanted a 99.7% confidence level we would need to multiply the standard error by three, expanding the interval to 100 ± 15.

A result is said to be *statistically significant* if the chance of observing that result by chance alone is considerably low for the chosen confidence level. What is a low chance in turn depends on the chosen confidence level — with a confidence level of 99%, for example, results as improbable as those which happen 1% of the time are considered statistically significant.

Aside from its central role in performance analysis, statistical methods are employed in various scenarios where it is not possible to know all the data of the population being analysed or when the measurements themselves pose some uncertainty due for instance to limits in the measuring apparatus. For instance, economics, risk analysis, weather forecasting, medicine and machine learning. Machine learning in turn is used by diverse fields within an environment where uncertainty is present and where the desired course of action is unknown a priori.

2.4 Machine Learning

Machine Learning is the study of algorithms which collect information and leverage that knowledge to improve the quality of their predictions [MRT12]. Machine learning is used to solve a broad set of problems which would otherwise require an

extremely large number of manually-crafted rules to decide, like natural language processing, face detection, speech recognition and document classification.

There are several different ways in which a machine learning algorithm may acquire data. With supervised learning, the learner receives labeled training data and, from that, must predict unlabeled points. In other words, the learner is provided with the correct action for some points and must use that knowledge to choose an action on new points it potentially has not seen before. While a common type of learning, its use is limited to situations where obtaining a general enough training set is feasible. In more interactive problems this is often not the case. Unsupervised learning, in turn, receives only unlabeled data and is used for instance in clustering. With online learning, there are multiple intermixed rounds of training and testing. Like with unsupervised learning the learner receives only unlabeled points, which it must label and, afterwards, learn the correct label for that point. Active learning is similar to supervised learning but instead of receiving all the training data at the beginning the learner asks an oracle for new points. The advantage of active learning is it can reduce the number of required points, which is an important trait in applications where the acquisition of labels is expensive like computational biology.

With reinforcement learning, the learner receives unlabeled points. Unlike unsupervised learning, whose aim is to find the underlying structure of the data, reinforcement learning is concerned not with the structure itself but with maximizing its reward. The reward can be anything like total energy consumed, peak power, execution time, etc. Reinforcement learning is often used in interactive environments, as opposed to receiving points that have been gathered previously. Compared to other types of learning, it has the additional challenge of having to handle the trade-off between exploration and exploitation [SB18]. To maximize its reward, the learner must execute actions which it knows that tend to produce good rewards. However, to know which actions tend to produce the best rewards it has to try new actions. Consequently, the learner cannot simply disregard exploration and exploit one action, as it may miss better rewards. Nor can the learner forego exploiting the best action and only explore the environment as doing that results in receiving many bad rewards as it fully explores the environment.

As an example of the use of reinforcement learning, let's take the (very simple) tic-tac-toe game. In this game two players compete against each other in a 3×3 grid. Players take turns placing an X, for one player, or an O, for the other in a free cell in the grid. Once there are three of the same symbol in a row, column or diagonal the game ends and the player who played that symbol wins. Alternatively, the game ends in a draw if the previously mentioned condition is not met and there are no more free cells in the grid. This game is solved, and by playing optimally it is impossible to lose. So, for this example, let's assume the adversary is not playing

optimally. That precludes the use of the minimax algorithm, as it would avoid reaching a game state that risks a defeat against an optimal player, even though our non-optimal opponent may make a mistake in that case and lose.

One way to solve the tic-tac-toe problem is with a value function. Each game state has a value, which is the probability of winning the game once that game state is reached. However, as we initially lack any knowledge of how the adversary plays, this probability is just an estimation. As games are played the values are updated and the estimation improved. As the learner is playing the game it must decide weather to act greedily, and attempt to reach states with high values, or exploratory, and attempt to reach states which still don't have a good estimate of how likely a victory is. There are different ways to balance exploration and exploitation, some of which are described later, but a simple solution would be for the learner to prefer exploration on its first games and, after some time, prefer exploiting its knowledge. Regardless of the exact means of balancing exploration and exploitation, if exploration tends to zero over time, this policy converges to optimality against any fixed opponent.

Uncertainty over the consequences of an action, lack of a mathematical model of the environment and the need to balance information gathering with making use of that information are common to a varied array of problems, such as playing games like backgammon or go, selecting the best product placement to increase sales, identifying new chess gambits, elevator scheduling, etc.

2.5 Autotuning

Many applications have the bulk of their computation done by relatively small pieces of code called computing kernels [Big+18a]. For example, sparse matrix-vector multiply in linear systems solvers, Fourier transforms in signal processing, and discrete cosine transforms in JPEG image compression [VD00]. Since it is a relatively minor portion of the code, improving the performance of these kernels, and consequently the application, should be easy to a degree. However, the computing kernel must be optimized for each target hardware architecture, resulting in not one but several repetitions of the already costly, time-consuming and error-prone process of manually tuning a computing kernel [Ben+14]. Autotuning attempts to reduce the programmer effort required for this performance tuning process by making performance gains more portable between architectures [Big+18b]. Autotuning is used both by individual applications and by libraries such as ATLAS [WPD01], FFTW [FJ98], MKL [Wan+14] and SPIRAL [Pus+05]. Some of these libraries provide several versions for the same computing kernel. In those cases, the choice

of version to use is made at runtime based on several parameters, including for instance the problem size.

An autotuner must do two things: code generation and performance evaluation. For code generation, usually, the developer provides the autotuning tool with code generators. This allows the autotuner to test many different values for each optimization, for example, many different tile sizes that would be cumbersome to write manually. Furthermore, the code generators enable the autotuner to automatically combine several optimizations, which is useful since even if each optimization in a set of optimizations is beneficial on its own when used together they may impact performance negatively. Performance evaluation entails running the generated code to measure how well it performs under some system configuration. However, it often happens the search space containing all possible versions that the code generators can produce for a kernel is far too large to be exhaustively explored. As the number of valid versions may be exponential to the number of parameters, even with a large amount of computational resources it may not be feasible to fully explore the search space regardless of the computational resources available. For example, the Poisson benchmark from PetaBricks autotuned has 10^{3657} possible configurations [Ans+14].

To illustrate the difficulties present in autotuning we will go through the autotuning process for a simple kernel. The kernel being autotuned is shown in Listing 5, and is a tiled matrix multiplication that computes $C = AB$. For simplicity, we will limit the performance tuning to three parameters: TILE_I, TILE_J and TILE_K. TILE_I will be shown in detail in the following paragraphs, while TILE_J and TILE_K are detailed in Appendix A. These parameters control the tile size used to access each of the matrices, changing the order in which cells are accessed, resulting in fewer or more cache hits on each of the matrices. Everything else is kept unchanged, all versions generated use the same compiler flags, values for loop unrolling, etc.

```
1  void mmul(int m, int n, int p, double *restrict A,
2  double *restrict B, double *restrict C) {
3    int i, j, k, ii, jj, kk, il, jl, kl;
4    for (ii=0; ii < m; ii += TILE_I) {
5      il = min(ii+TILE_I, m);
6      for (jj=0; jj < n; jj += TILE_J) {
7        jl = min(jj+TILE_J, n);
8        for (kk=0; kk < p; kk += TILE_K) {
9          kl = min(kk+TILE_K, p);
10         for (i=ii; i < il; i++)
11           for (j=jj; j < jl; j++)
12             for (k = kk; k < kl; k++)
13               C(i,j) += A(i,k)*B(k,j);
14 }}}}
```

Listing 5: Example of a tiled matrix multiplication kernel.

The effect of changing the tile size in the order the matrices are accessed is shown in Figure 2.6, which depicts a multiplication of two very small 4×4 matrices. In the figure, TILE_I and TILE_J were kept constant at 2 and 4, respectively. Each column corresponds to a different value for TILE_K, varying between 1 and 4. The access pattern used by the matrix A is shown in the first row and the pattern used for B, with the same tile sizes, is shown in the second row. In each pattern, each point represents a memory access, with the vertical axis representing the matrix row and the horizontal axis the matrix column being accessed. Notice that each of the points shown is accessed 4 times. Furthermore, notice it is not possible to use the best pattern for all matrices. For instance, the pattern for A shown in the fourth column of the first row, which accesses adjacent columns sequentially before advancing to the next row, can only be used together with the pattern for B shown in the fourth column of the second row, which does the opposite, accessing adjacent rows before advancing to the next column. When reading an element of the matrix, first this element is searched in the CPU cache. If it is not found in the cache, the element and those adjacent to it in memory are loaded from memory and stored in the cache. This improves memory accesses when data is accessed sequentially, as much of the data will be cached by the time it is accessed. As such, in Fortran it is more efficient to access, sequentially, elements adjacent by row as these are adjacent in memory. In C/C++, it is the opposite and it is more efficient to access, in sequence, elements adjacent by column. Regardless of how the data is distributed in memory, we can see that, with this kernel, if we improve the accesses to A we may be worsening the accesses to B. Consequently, from the performance point-of-view, there are no trivial values for the tile sizes that are clearly better than others tile sizes.

In this example, we use a 2048×2048 input matrix. This means there are 2048 possible values for each of the tiles, the combination of which results in $2048^3 \approx 8.5$ billion possibilities in total. Furthermore, each of these must be executed more than once in order to reduce the effects of the environment on the measurements. As we chose to execute each version 5 times, an exhaustive search would require over 40 billion executions. Finally, there is also the time required to compile each version, which in the example is small, of about half a second. As an exhaustive search would take a very long time to finish (about 1000 years, using the average execution time of 7.9 seconds found in the experiments), the first step is to limit the search space from all possible values to only those values that are reasonable, given what we know about the kernel and the target system.

First, we execute a small experiment using only a few powers of two within the ranges of each optimization parameter. More specifically we used, for each parameter, the values 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, and 1024. As we have 11 possible values for each parameter, this leads to $11^3 \times 5 = 6655$ executions in total. Inspecting

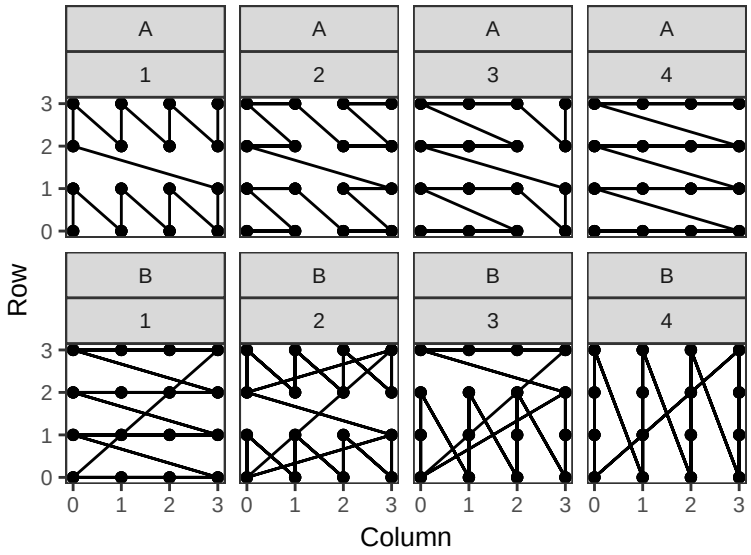

Figure 2.6.: Access patterns when multiplying two small 4×4 matrices, using 2 for TILE_I, 4 for TILE_J and varying TILE_K between 1 and 4, shown in gray. The first row shows the different patterns for A, with the access pattern for B using the same tile sizes shown immediately below on the second row. The horizontal axis shows which column is accessed and the vertical axis shows the respective row.

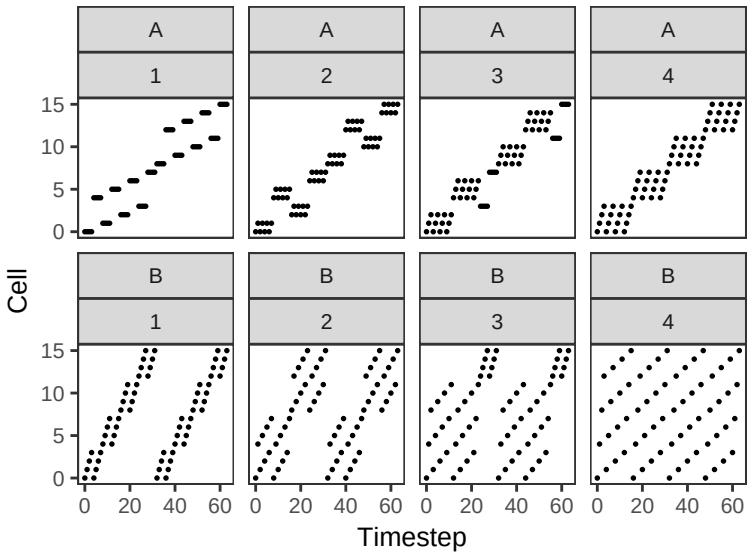

Figure 2.7.: Access patterns when multiplying two small 4×4 matrices, using 2 for TILE_I, 4 for TILE_J and varying TILE_K between 1 and 4, shown in gray. The first row shows the different patterns for A, with the access pattern for B using the same tile sizes shown immediately below on the second row. The horizontal axis shows the access order and the vertical axis shows the cell accessed, with the cell number given by $4 \cdot \text{row} + \text{column}$.

even just three parameters together can be very difficult, so we will analyze TILE_I, TILE_J, and TILE_K separately.

Figure 2.8 shows the effect on the execution time of changing the value of TILE_I. The vertical axis shows the execution time, in seconds. The horizontal axis shows the value used for TILE_I. Each point is the minimum execution time of all executions where TILE_I has the value shown in the horizontal axis. That is, the figure shows how TILE_I affects the execution time when using the best known values for the other two parameters. We can see TILE_I affects execution time and can make it vary between 1.8 second at 4 and 2.2 seconds at 256. Furthermore, the curve at the left of the figure indicates the best value for TILE_I lies between 1 and 8, although there is no guarantee there is not some other value of TILE_I elsewhere would not result in better performance.

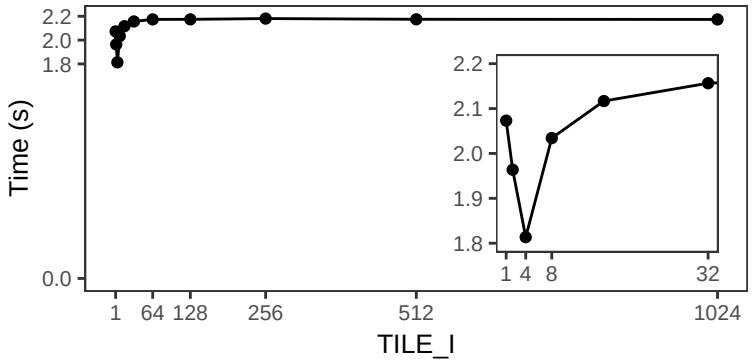

Figure 2.8.: Effect of different values of TILE_I on the execution time. Each point, one for every value of TILE_I, represents the execution time using that value of TILE_I and the values of TILE_J and TILE_K which give the lowest execution time for that TILE_I.

At this point of the autotuning the best point takes 1.8 second to execute and uses 4, 256, and 1 for TILE_I, TILE_J, and TILE_K, respectively. There are often diminishing returns, as we approach the ideal values of the parameters, so we could opt to stop autotuning at this point. Alternatively, we could use the knowledge we gathered so far to attempt to find more fine-grained values for the parameters. In this case we will continue autotuning by exploring points in the ranges 1–8, for TILE_I, 64–512, for TILE_J, and 1–32 for TILE_K.

The results of these new experiments are shown in Figures 2.9. The figure shows the effect on performance of varying TILE_I. In the previous step the best value for this parameter was 4. We can see 5 and 6 result in shorter execution times. However, this difference is very small: 1.813 second for 4, 1.793 second for 5, and 1.796 for 6, i.e. a difference of 1.1% between 4 and 5. Furthermore, we have only five executions for each. To confirm there really is a difference, albeit small, between using these values for TILE_I we executed each of them 50 times, as the 5 executions we have thus far do not allow us to tell whether they differ from each other or if the measured difference is only because of external factors. These results are shown in Figure 2.10. In that figure each point is the average of 50 executions, and

the bars around each point show its confidence interval at a confidence level of 99.7%. Although points 4 and 5 are very close, their intervals are far apart enough that there is a difference between the two. We can therefore expect to gain 1.4% of performance by using a value of 5 for TILE_I compared to using the previous known best value of 4.

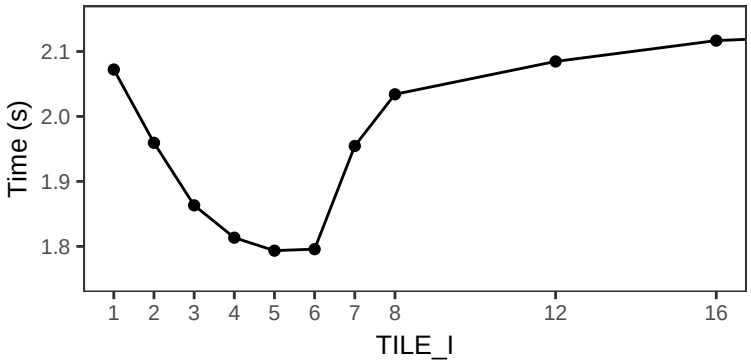

Figure 2.9.: Effect of different values of TILE_I on the execution time. Each point, one for every value of TILE_I, represents the execution time using that value of TILE_I and the values of TILE_J and TILE_K which give the lowest execution time for that TILE_I.

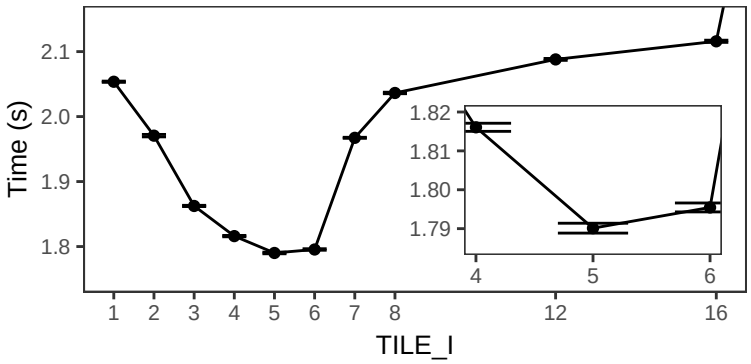

Figure 2.10.: Effect of different values of TILE_I on the execution time. Each point, one for every value of TILE_I, represents the execution time using that value of TILE_I and the values of TILE_J and TILE_K which give the lowest execution time for that TILE_I. Each point is the average of 50 executions and the bars indicate its confidence interval at a 99.7% confidence level.

One could opt to search the rest of the configuration space for better values for TILE_I, or further expand the search space by adding other parameters like compiler flags. However, at this point we opted to stop the autotuning process as the performance gains of the previous step were very small. Notice this example was not just very small, with a single kernel and only three parameters, but targetted a single platform. Real applications have a larger number of computing kernels to be tuned, often with more than three parameters each, and the autotuning has to be repeated for each target architecture, making it impractical to do this tuning manually.

One alternative to make feasible the search in a large configuration space is to prune large swaths of configurations. This, however, risks accidentally pruning optimal solutions. To mitigate this risk, machine learning is often employed instead of directly pruning the search space. There are many machine learning methods which can be used, and the best search technique depends on the configuration space. As such it can be advantageous to use more than one search technique. For example, OpenTuner [Ans+14] implements differential evolution, several variants of the Nelder-Mead method and Torczon hillclimbers, evolutionary algorithms, particle swarm optimization and random search; BOAST [Vid+18] implements an evolutionary algorithm and random search. These techniques are then employed through a meta-technique named the multi-armed bandit with sliding window, area under the curve credit assignment, a variant of the multi-armed bandit problem described next.

2.6 The Multi-armed Bandit Problem

In autotuning there is, obviously, a limit on how many computational resources and how much time the autotuner is able to employ on a given problem. The autotuner must, then, allocate its limited resources to learn more about the different optimizations and, with that, infer which of the kernel's versions provides the best performance. Multi-armed bandit problems share some similarity with the autotuning problem. Both have a finite budget of resources, and must trade them for knowledge. The major difference between the two is that the sole task of the autotuner is to identify which version performs the best, a bandit problem however does not have this as a primary, but as a secondary objective, its primary objective being maximizing its reward, or, using the autotuning scenario, minimizing the total execution time of all the kernel versions executed while still executing the same number of those.

Multi-armed bandit problems [Rob52] are so named because one of the early experiments covering this kind of problem used a two-armed *bandit* [LS18]. One-armed bandit is a name for slot machines, which have a single arm and "rob" the gambler due to the odds favouring the house. In the two-armed bandit experiment, each arm had a different stochastic distribution for its reward. These distributions were not known by the gambler, who had to test both arms as they attempt to maximize their payout. Multi-armed bandits can model a broad range of problems found in vastly different domains, like testing pharmaceutical treatments whose efficacies are not yet well understood, and distributing funds to several research projects which do not have well-defined payouts.

In the general case a K-armed stochastic bandit possesses not two but K arms, each described by some distribution $P_i \forall i \in \{1, \ldots, K\}$. At every round t the gambler chooses some arm $A_t \in \{1, \ldots, K\}$ and pulls it, receiving some reward $X_t \sim P_{t,A_t}$. This process continues for n turns if the event horizon is finite, or indefinitely case contrary. As the gambler cannot directly observe Pi the decision of which arm to choose at round t depends solely on $A_i \forall i \in \{1, \ldots, t-1\}$, $X_i \forall i \in \{1, \ldots, t-1\}$ and, if known, n in order to maximize its total reward $S_n = \sum_{t=1}^{n} X_t$ at the horizon.

One way to evaluate how well one policy performs is to use *regret*. The regret of a policy is the difference between its total reward and that of the optimal policy. That is, the farther a policy is from the optimal the higher its regret. Accordingly, for a K-armed stochastic bandit ν and with $\mu^*(\nu) = max_{i \in [K]} \mu_i(\nu)$ the largest mean among the arms. The regret of a policy π in a bandit ν is given by

$$R_n(\pi, \nu) = n\mu^* - \mathbb{E}[\sum_{t=1}^{n} X_t]$$

One strategy to find the arm with the best average payout is the Explore-then-Commit algorithm. It explores (pulls) each arm a certain number of times and, from then on, only uses one arm. The Upper Confidence Bound algorithm chooses each arm once, then it chooses the arm which would be the best being optimistic regarding the data observed so far. As the name indicates, this is done by choosing the arm with the highest upper confidence bound, which is an overestimation of the mean.

2.7 Concluding Remarks

In this chapter we briefly described multicore architectures. We explain how unlike other parallel techniques like ILP the use of multicore is not transparent to the programmer and requires conscious programming effort to take advantage of the multiple computational units. We succinctly present the further difficulties present in NUMA systems, which require careful placement and access of data in the right nodes to avoid for example reaching the bandwidth limit of one node while another remains idle. We detail some parallel programming models and implementations, namely POSIX Threads, OpenMP, OMPSs, StarPU and Kaapi. We briefly compare two OpenMP runtimes, libGOMP, used by GCC, and libOMP, which is used by LLVM and is the runtime our software artefact was based on. Furthermore, we present some statistical techniques which can aid in data analysis and which are also used by the version selector used by our proposal.

We aim to provide better performance for multithreaded applications under different scenarios without sacrificing performance portability. In order to do this we use autotuning to automatically generate the alternative versions of a kernel, using machine learning to decide which kernel version to use in a given scenario during execution within the LLVM OpenMP runtime.

3 Machine Learning Assisted Autotuning of Multi-Versioned OpenMP Tasks

Chapter 2 highlighted some of the challenges inherent in improving the performance of an application. Many applications spend the bulk of their execution time in a few computing kernels. For instance, the seismic-wave propagation simulator Ondes3D [Dup+09; Dup+08] spends most of its time on just a few kernels for computing and updating parameters like the velocity and stress of the seismic wave. Accordingly, an important part of performance-tuning these applications consists of performance-tuning, individually, each of the computing kernels which comprise the application.

Typically, optimizing a kernel consists of identifying possible optimization points in that kernel's code; implementing optimizations which the developer believes should improve performance in a target platform, for instance through the use of vectorial instructions or reducing memory accesses by making better use of caching; benchmarking each of the optimizations on the platform, including combinations of optimizations when possible. With the performance of each optimization known, for each kernel, the application is then finally compiled to make use of the best known version of those kernels for the target platform. This process can, however, only find the set of optimizations which are generally the best for the benchmarking data. That means a bad benchmark may cause the selection of a set of optimizations that are sub-optimal when used with the real data.

To further complicate the optimization process described above, the optimal version of a kernel may be input-sensitive. That is, the optimal version may be different depending on some property of the input. For example, to compute the minimum spanning tree (the minimum set of edges that keeps the graph connected) of a weighted graph the algorithm Prim [Pri57; Jar30] (which has a complexity of $O(E + V \cdot log(V))$, where E is the number of edges and V the number of vertices), making use of Fibonacci heaps, often outperforms the Kruskal's algorithm [Kru56] ($O(E \cdot log(E))$). However, if the input graph is sparse, that is, if the graph has many vertices relative to its number of edges, as is the case for example of social network or road graphs, the second algorithm tends to outperform the first. A more

concrete example is the *std::stable_sort* function of the GNU implementation of the C++ Standard Template Library (STL). This function sorts the input maintaining the order between elements with equal keys, such that if one first sorts the data using a key k_0 and, subsequently, sorts the result using a key k_1 the resulting data will have items sharing the same k_1 ordered by k_0. The *std::stable_sort* function uses two different sorting algorithms: insertion sort when the input has fewer than 15 elements; otherwise mergesort is used, splitting the input in two and calling the function recursively. Tuning an application to exploit input properties requires not only including in the application the different ways to perform the computation, but makes it necessary to check the input before passing it to the kernel version that will handle it, as is the case of *std::stable_sort* checking the number of elements before deciding how they will be sorted.

Finally, there is the issue of optimizing for multiple platforms instead of a single platform. Optimizations that are beneficial in one platform may be disadvantageous in another [Tri+03]. For example, loop unrolling may improve performance as it can increase instruction level parallelism. However, it may also be detrimental to performance as it increases the memory required by the code, which can result in increased cache misses. Moreover, the optimizations that perform well with a particular input may be harmful with another. In the case of *std::stable_sort* for instance the cutoff point of 15 elements was generally good for the architectures of 2001, when the code was written. However the value is not adequate for modern architectures [Din+15]. Despite this, as the library is commonly distributed in a compiled form and with a static value for the cutoff, this performance optimization is not portable to modern architectures. Better performance with new architectures could be obtained by applying the whole optimization process for each of the target platforms, however this is often impractical, especially when the code in question is used for several years and is run in architectures which did not exist at the time of the initial optimization phase.

In this chapter we present our approach to provide the developer with better tools to tackle some of the difficulties in performance-tuning a task-parallel application. We present our proposal in Section 3.1. Section 3.2 explains the algorithms we use for deciding which version to use for a task. Section 3.3 details how we integrated our approach in OpenMP. Lastly, Section 3.3 concludes this chapter.

3.1 A New Approach For Autotuning

This section presents our proposed solution to improve the performance of parallel programs using the OpenMP runtime. An OpenMP task is composed of data and a

piece of code [LAC]. We extend the OpenMP task construct so that it has multiple, functionally equivalent, versions of this code. The reason for this change is to permit the use of task versions with different performance profiles, enabling the application to use a different version according to the circumstances. For example, task versions can be the same code compiled with different compiler flags [GS16; HE08; NMC15]. Versions can differ more than by compile-time flags, changing parameters like tile sizes, allowing the use of the same application executable with different platforms while optimizing the execution for each of their cache sizes. Task versions can differ even further. For example, by using different algorithms for a task which sorts an array, a task could use insertion sort for small inputs and quicksort for larger inputs [Sed78; LGP05].

While having multiple versions can improve performance, this only happens if the correct version is used. One possibility is for the user to explicitly define the situations when each version should be used. However, that approach demands a great deal of effort from the developer, as it requires extensive benchmarking to identify all situations where each task version should be used. Another disadvantage is that it restrains the optimizations to situations that have been seen before, requiring further training as new situations emerge or new platforms are targetted since code autotuned for one platform is not guaranteed to perform well on other platforms [SRD16]. Instead of statically defining when each task version should be used we opted for dynamically learning what is the best task version. Note that there can be more than one best version for a platform.

In early computer architectures, the task version which executed the fewest instructions would, usually, be the most efficient. However, due to the complexity of modern computer architectures, it is often unknown which version has the best performance in some target architecture before all versions have been executed. In order to search for the best version we need to know the size of the problem being solved. This is necessary so we can tell whether some version finished its execution quicker than another version because it is indeed faster or if it is because the solved problem was smaller. For example, the naive matrix multiplication algorithm has a time complexity of $O(n^3)$, but if it is executed with a problem of size 2 it should, theoretically, take less time than the more efficient Strassen's algorithm ($O(n^{\log_2 7})$) with a (larger) problem of size 4 (ignoring constants and other factors in this example). That is to say, in that case, the naive version would finish faster — but only because it solved a smaller problem than the other version did. Knowing the problem size also has the benefit of allowing the use of a different version depending on the input. This provides an advantage over using autotuning alone, as autotuners typically find a single optimized version per platform, regardless of input. Yet many problems can benefit from being input-aware [Din+15]. It is the case of many BLAS operations, for example, as evidenced by many BLAS

libraries using one version for large and another for small matrices. Note that our proposal is not opposed to autotuning, as it can leverage autotuning to generate the multiple versions, which can be useful for instance to reduce the number of versions available to a manageable number as an autotuner can handle a larger number of versions than we can during execution.

The input size, by itself, however, may not provide the actual computational cost of the task. Some algorithms have a very predictable computational cost. The multiplication of two matrices, for example, ignoring environmental factors like data locality and dynamic CPU frequency, has a very predictable execution time based on the input size. That is not the case, for instance, of a heap search, where the time required depends on the input size, but it also depends on whether the element being searched is close to the root of the heap or its leaves. In those cases it is often impossible to know how difficult the problem being solved actually is before it is solved. In that scenario one has to rely on the asymptotic cost, requiring more executions of the task version with a certain input size before its performance can be known with some degree of certainty.

The multiple task versions require no changes to the way tasks are scheduled. However, they add a subproblem to the scheduler: with a single version the only job of the scheduler is to decide when and where a task should run, with multiple versions the scheduler must also decide which version will be used for the task. Ideally, the scheduler would always run the best version for the current scenario (problem size, data properties, data locality, processor occupancy, etc). However, the scheduler does not know which is the best version, so, instead, whenever a task is launched it must choose between using the version that, based on the information it has gathered so far, is most likely to be the best, or run some other version in the hopes that one of them is better than it seemed at first. This decision is made by the selection policy, which is discussed in more detail in Section 3.2.

Figure 3.1 shows how our extension affects the compilation process, with our additions shown in blue. The leftmost part of the figure shows the multiple computing kernels used in the source code. In this part, the blue boxes represent the possibility to have multiple versions of each kernel. The rightmost part of the figure shows the resulting application, which contains all kernel versions from the source code. We made these additions to the LLVM [LA04] C/C++ compiler (clang). More details are provided in Section 3.3.2.

Figure 3.2 shows how our extension affects the OpenMP runtime, during the execution of the application. The leftmost part of the figure shows the directed acyclic graph containing the tasks and their dependencies. When the task scheduler decides which task to run, that task is executed in one of the available workers,

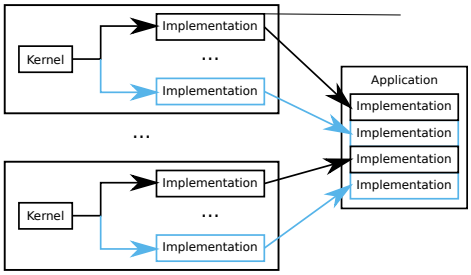

Figure 3.1.: Overview of how our extension interacts with the compiler. Items in black show current compilers and runtimes; items in blue show our additions.

illustrated on the rightmost part of the figure. With our additions, however, before the execution can begin the runtime must first find which versions are available for that task. Once the versions are found, the runtime decides on a version to run based on data collected during previous executions. Once the task finishes executing the collected data is updated with the timing information of that execution and the worker used for that execution is available to execute another task. Section 3.3.3 provides more details on the changes made to the LLVM OpenMP runtime.

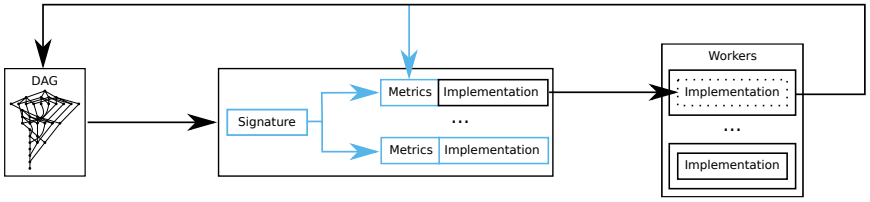

Figure 3.2.: Overview of how our extension interacts with the OpenMP runtime during execution of the application. Items in black show current compilers and runtimes; items in blue show our additions.

3.2 Task Version Selection Policies

With the use of multiple versions per task, the scheduler must select which of the versions versions to run — further adding to the challenge of selecting a task. Some approaches, like some of the policies of StarPU [Aug+11] and OmpSs [Dur+11], rely on past executions of the application to help guide the scheduler to make better decisions instead of having to discover the same information every time the same application is executed. We have opted however for not depending on past executions, relying only on the information obtained during a single execution of the application. We made no changes to OpenMP's scheduling algorithms. Consequently, the order in which tasks are executed is maintained aside from side effects like changing the task execution order due to changes in task execution times, affecting which instructions fit in the instruction cache due to changes in code size, etc.

To minimize the execution time, the runtime should, evidently, always use the fastest available version of a task. However, initially the runtime is unaware of each version's performance. As such, the runtime must also identify which among all the versions is the fastest. That is, the runtime has two responsabilities: exploration, which consists of measuring the performance of all the versions, and exploitation, which consists of running the fastest version as often as possible.

If distinct executions of a kernel version always had the same execution time, this task could be easily accomplished by executing each version once, and, from that point onward, always use the fastest of those. That is unrealistic — the same code, with the same input, will exhibit different execution times if run several times. This happens due to all the complexities present in a modern computer system. There are other applications running simultaneously, which use shared resources like CPU cores, cache, and memory. One execution may have all its data available in cache, another may have to wait to load it from the memory. Furthermore, the processor states will be different from one run to another, for instance due to temperature changes one run may be able to use a higher frequency than another [Cha+09], or the contents of the shared cache may be different. Due to this imprecision present in the execution times of a kernel version, relying on the execution time of a single execution could easily lead to a sub-optimal choice of kernel version by a naive algorithm. Which, in turn, could drive the algorithm to perform poorly.

It follows that the version selection policy must execute at least some of the available kernel versions more than once. The question then is how many times and when should each version be executed, as simply identifying which of the kernel versions performs the best is insufficient, the algorithm must also make use of that information to improve the execution time. This problem is further complicated as the horizon, or the number of times a task must be executed, is only known once the application finishes executing. During execution this information is not generally known as a task may spawn new tasks. As the size of the input data of a task tends to strongly affect execution time, we make the version selection independent between different input sizes.

Every time a task is scheduled to begin execution the runtime may take one of two possible courses of action. It may execute the version which performed best up to that point in the execution. This course of action is considered greedy as it focuses on immediate gains. The runtime might, instead, opt to further explore one of the versions that have, so far, shown worse performance. Executing one of these other versions may worsen the execution time as they have been slower in the past. However, it is also possible the real best version is not the one that has been the fastest so far — which is possible as external factors such as data being in the cache or not, or different amount of memory bandwidth available influence the measured

execution time. These two objectives are antagonistic, however both exploration and exploitation are crucial for improving performance.

3.2.1 Task Versions as a Multi-Armed Bandit

The problem of choosing which task version to run can be modeled as a stochastic multi-armed bandit, which is described in greater detail in Section 2.6. The problem consists of minimizing the total execution time of n executions of some task T. This task can be executed using any of k versions, each with a possibly different performance profile. There are as many bandit problems to solve as there are tasks (or, equivalently, it is an associative bandit), and each of those bandits is considered independent from the others. The k versions are represented in the model as the arms of the bandit. Each execution of a task is expressed as a turn in the model. When the t-th task is to be executed the runtime must choose a version to perform the execution. This is modelled as the learner choosing some action $A_t \in \{1, \ldots, k\}, \forall t \in \{1, \ldots, n\}$ which defines which arm to pull. We want to minimize the execution time, or, more formally, the objective is to minimize the regret [GLK16b] given by

$$\mathbb{E}_\mu \left[\sum_{i=1}^{n} \mu_{A_i} \right] - n \cdot \mu_{\text{argmin}_{i \in 1 \ldots k}(\mu_i)}$$

where μ_i is the mean of the i-th arm. Note that if only the optimal arm is pulled the regret is zero, however the learner does not know which is the optimal arm.

In the bandit model, at turn t the learner knows the reward provided by all previous pulls, given by $X_i \forall i \in \{1, \ldots, n\}$. However in the actual problem being modeled as a bandit problem there is no guarantee the knowledge of any previous X_i will be available at the moment the t-th task begins execution. This is because the model we use presumes the pulls are made in sequence, and the reward provided by pulling an arm is available as soon as that arm is pulled. In the version selection problem, however, the task version requires some time to execute. Consequently, the reward of each pull is only known when that task finishes execution. Since another task of the same type may start execution before this task finishes, it is not possible to guarantee X_{t-1} is known at the t-th pull. Even delaying the execution of the t-th task to after all $X_i \forall i \in \{1, \ldots, t-1\}$ are known, which would have a very negative effect on performance, would not solve the issue as a task may launch subtasks of the same type. For example, a task t may launch a subtask $t+1$ and only finish once $t+1$ finishes, meaning we cannot require $t+1$ to wait on t without creating a deadlock. Therefore, we opted to use the knowledge available at the time the task version is being selected, even if this knowledge is limited. Accordingly, X_{t-1} is known not at turns $i, \forall i \in \{t, \ldots, n\}$ but only for the turns after that task finished

executing. Another consequence of this design decision is that X_i may be known and X_j unknown even if $j > i$. Each arm, when pulled, provides the learner with a reward $X_i \sim N(\mu, \sigma^2), \forall i \in \{1, \ldots, k\}$, with an expected value of μ_i. That is, each arm has a Gaussian-distributed reward with some initially unknown mean and variance. Additionally, these distributions are independent from one another, such that pulling one arm does not affect the distributions of the other arms.

There are many approaches for solving a bandit problem [SB18]. Greedy policies estimate the value of each arm and simply use the arm with the highest expected value. These policies do not value exploring the environment, opting instead for maximizing the current total reward by exploiting the arm that, to their knowledge, should provide the best reward. These policies have an advantage when the reward distributions have very small standard deviation, as the sampled values in that case are likely to be close to the population's average. The disadvantage of greedy policies is they incur a very large risk of choosing a suboptimal arm and, due to the lack of exploration, continue to choose the suboptimal arm every turn, resulting in a large regret. In our scenario these policies can differentiate between task versions when these have vastly different performance (for instance, when choosing between a naive matrix multiplication algorithm and Strassen). However, they cannot reliably detect small differences in performance, making them unsuitable for comparing task versions which are just slightly different, as is the case between many tasks generated with autotuning. A variation of this class of policies is Greedy With Optimistic Initialization, which sets the initial estimates to a high reward, in turn making it explore the arms for longer. ϵ-Greedy policies usually operate greedily, pulling the arm with the highest expected reward. However, every turn there is a small probability (ϵ) one of the other arms will be pulled instead. This guarantees the method eventually finds the best arm, regardless of the probability, if there is a large enough number of turns. There is of course a trade off on the exploration probability, higher values mean the best arm will be identified earlier, enabling the greedy use of that arm sooner. However, the greater the exploration probability the less likely the method is to take the greedy action. Some ϵ-Greedy policies opt to start with a large ϵ and decrease it gradually in order to spend more time exploring in the beginning and more time exploiting later in the execution. Confidence-bound policies do not have separate explore and exploit phases. Instead, these policies have a confidence bound for each arm. At each turn, the arm whose confidence bound is the largest is selected[1]. At the beginning of the execution these bounds overestimate the mean, and every time an arm is pulled its confidence bound is brought closer to the mean. This allows the confidence-bound

[1] In our use-case we pull the arm with the lowest confidence bound instead of the largest, as we want to minimize the execution time. The name of the confidence-bound policy we use however implies the largest bound is used since the algorithm describes a maximization problem. It performs equally well for minimization, with trivial changes in its algorithm.

policies to do more exploration in the beginning and focus on exploitation as more information is gathered.

The following subsections explain in detail the policies we use and the adaptations necessary for our use-case. Subsection 3.2.2 presents the greedy policy Mean. Subsection 3.2.3 describes the Upper Confidence Bound policy. Subsection 3.2.4 presents the Gradient Bandit policy, which makes use of randomization for its decisions.

3.2.2 Greedy Policy Mean

The Mean policy is perhaps the most straightforward way to handle a bandit problem. It uses a Explore-Then-Commit (ETC) strategy, whose name concisely describes its simple approach: it first explores the environment and then it commits to a single task version. The exploratory phase consists of running each task version successively in a round-robin fashion. Each of the k versions is pulled an arbitrary number of times, m. As a consequence, the exploration phase lasts for $m \cdot k$ rounds, regardless of the distributions followed by the execution times of the task versions. After this exploratory phase is over, the policy always uses whichever version was the fastest so far. Since the horizon n, which is the total number of times the arms must be pulled, is unknown, one cannot know beforehand if some value of m is too large or too small. When m is too large, the policy will spend too long learning the distributions, limiting how much it is able to make use of the gathered knowledge. If, on the other hand, m is too small for the horizon, the strategy will make more use of the gathered knowledge but with a high risk of mistakenly concluding a suboptimal task version is optimal, in turn running several times a version with suboptimal performance and causing overall performance to degrade. Were the horizon n known, m could be chosen optimally. However, even in the simplest case, with two versions, a known horizon, and a value of m chosen optimally, any ETC strategy is still suboptimal [GLK16b], with a lower bound of $\frac{log(n)}{|\mu_1 - \mu_2|}$ for its regret. In many cases it is unfeasible to choose m optimally as the horizon is unknown, in which case ETC cannot guarantee even that lower bound. For our use-case, in order to choose m optimally, the user would need to know, when launching the application, the number of times each task type is launched and with which data sizes, which is unfeasible in some applications.

The ETC algorithm is shown in Listing 6. Exploration is done in line 3, for $k \cdot m$ rounds. The commitment phase is executed by line 5. Notice we use argmin instead of argmax as we want to minimize, not maximize, the execution time, whereas often the objective in a bandit problem is to maximize the reward.

```
1 function ETC(k, t, m) {
2   if t ≤ k · m {
3     A_t = (t mod k) + 1
4   }else{
5     A_t = argmin_i μ_i(t)
6   }
7 }
```

Listing 6: Explore-Then-Commit algorithm: k is the number of task versions; t is how many times the task has been executed so far; m is the parameter defining how many times each version should be sampled before commiting; A_t is the action taken at the t–th round, that is, the version chosen to run at that round; $\mu_i(t)$ holds the average execution time of the i-th task version after the task has been executed t times.

1	Round	1	2	3	4	5	6	7	8	9	10	...	100
2	Version executed	A	A	A	A	A	B	B	B	B	B	...	B
3	Time (ms)	899	1104	909	1013	975	972	958	749	1051	777		729
4	**A**												
5	Avg. (ms)		1002	971	981	980							980
6	Std. Dev.		145	116	97	84							84
7	**B**												
8	Avg. (ms)							965	893	933	901		903
9	Std. Dev.							10	125	129	132		100

Table 3.1.: Example execution of the Mean policy. Each column shows the data for one round. Line 1 shows the round of the column; Line 2 shows the version executed for that round, A for the first 5 rounds, and B afterwards; Line 3 shows the time the chosen task version took in that round; Lines 5 and 6 show the average execution time and standard deviation of A up to that point in the execution, the same statistics for B are shown in lines 8 and 9.

The deterministic nature of this policy makes it an ineffective policy for adversarial bandits [Aue+95], which is a variation of the problem where the environment knows the policy used and aims to minimize the reward. However, those are out of the scope of this thesis.

To illustrate how this policy behaves in a simple scenario, let's say we have two versions of a kernel, A and B. These two versions are illustrated in Figure 3.3, with A taking an average of 1 second to execute and B taking an average of 0.9, and both having the same standard deviation of 0.1 second. This policy begins by executing A, and then B, five times each, as shown in line 2 of Table 3.1. After executing A five times the policy calculates its average execution time, 980 ms, shown in line 5. The same happens after executing B, which after five executions has an average execution time of 901 ms, shown in line 8. From then on the version whose average is lowest is always executed, in this case B. If for some reason B started to perform worse, and its average time became larger than the average time of A, the policy would start to execute A instead.

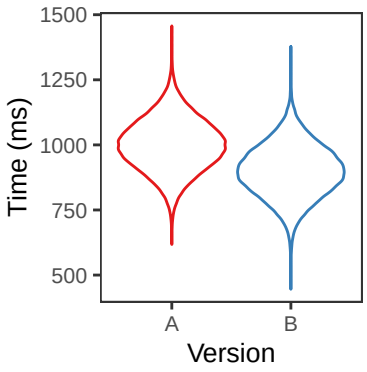

Figure 3.3.: Example of two the density distributions of two versions of a kernel.

3.2.3 Upper Confidence Bound (UCB)

The Mean policy attempts to maximize its reward by fully exploiting its acquired knowledge after an initial exploration. The Upper Confidence Bound (UCB) [Agr95; ACF02] policy uses a more balanced approach: it never fully commits to any task version, leaving always open the possibility of further exploring the other versions even if these have a lower mean reward. This behaviour is justified in a stochastic environment as the true distributions of the versions are never known — even after a very long execution there is still uncertainty, however small. Unlike the previous policy, with a sufficiently long execution UCB achieves a uniform logarithmic regret even with no previous knowledge of the reward distributions.

In contrast to ETC strategies, UCB does not explore based solely on the number of times a task version has been executed. Instead, it follows the optimism in the face of uncertainty principle. This principle states that actions should be chosen assuming the environment is as good as reasonably possible, from the point of view of maximizing the reward. The "reasonably possible" part of this principle is a parameter which can be used to make the algorithm behave more or less optimistic. By being less optimistic, the algorithm will be more conservative when selecting task versions to run, concentrating more on the exploitation of the gathered knowledge. However, that comes at the price of increasing the odds of exploiting a sub-optimal task version. In contrast, if the algorithm is too much optimistic, it will spend most of its time exploring the different options, gathering more information but missing many exploitation opportunities.

The first actions of UCB are the same as those of the Mean policy: it begins by exploring the different versions. With this initial data, the algorithm computes an upper confidence bound for each version. The confidence bound estimates how large the true mean execution time of the task version can be expected to be, based on the data gathered thus far and on the optimism level. The bound however

does not directly estimates the true mean, but, instead, provides an interval inside which there is a high probability the true mean is found. This is an important distinction. As the bounds are first built using the little information available at the time beginning, and they must include the mean with a high probability, these bounds must also initially overestimate the true mean. The overestimation however must still be reasonable. For example, a bound of infinity would be guaranteed to contain the true mean. However, such a bound would provide no information on the actual value of the mean. A confidence bound closer to the true mean, even if it isn't guaranteed to be correct, provides more useful and actionable information. UCB makes use of this fact by always choosing the arm whose upper confidence bound is the largest. Consequently, versions whose bounds are far from the true mean will be pulled, and their bounds, consequently, tightened and brought closer to the true mean.

There are different ways to estimate the upper confidence bound, chosen according to the assumptions made on the distribution. Too tight bounds will often underestimate the true mean, preventing the algorithm from working as intended. On the other hand, too loose bounds will cause the algorithm to spend too much time exploring instead of exploiting its knowledge. In our case the upper confidence bound (actually the lower bound, since we want to minimize, not maximize, the execution time) of a task version i at time t is given by [ACF02]:

$$UCB_i(t) = \overline{x}_i - \sqrt{k\frac{q_i - n_i\overline{x}_i{}^2}{n_i - 1}\frac{ln(t-1)}{n_i}}$$

Where: k controls the degree of optimism, $\overline{x}_i$ is the average execution time of version i after $t-1$ tasks of that type have been launched (this only includes execution times of tasks which have finished, not those still in execution), q_i is the sum of squared execution times, n_i is the number of times the task version i has finished executing. Some other possible estimates for the upper confidence bound are $\overline{x}_i\sqrt{\frac{2log(t)}{n_i}}$ [ACF02], $\overline{x}_i - c\sqrt{\frac{\ln t}{n_i}}$ [SB18], and $\overline{x}_i - \sqrt{\frac{2\log(t)}{n_i}}$ [Lai87; GLK16a].

The UCB algorithm is shown in Listing 7. Lines 2-3 guarantee each version is used at least once. Line 5 selects, for round t, the version which should provide the best reward. In this line the bound of version i at round t is estimated as:

$$\overline{x}_i - \sqrt{k\frac{q_i - n_i\overline{x}_i{}^2}{n_i - 1}\frac{ln(t-1)}{n_i}}$$

.

To show how this policy behaves in a simple scenario, we will use it in a simple situation where there are two versions of a kernel, A and B, with their execution

```
function UCB(v, t, k) {
  if t ≤ v {
    A_t = t
  }else{
    A_t = argmin_i x̄_i − sqrt( k (q_i − n_i x̄_i^2)/(n_i − 1) · ln(t−1)/n_i )
  }
}
```

Listing 7: UCB algorithm. v is the number of versions available for the task; t is the number of tasks of this type that have been launched, regardless of the version used; k controls the degree of optimism; A_t is the version chosen for the t-th execution of the task; $\overline{x}_i$ is the average time of task version i after t tasks of this type have been launched; q_i is the sum of squared times of version i; and n_i is the number of times version i has been executed.

times shown in Figure 3.3. Table 3.2 shows the first 10, and the last, steps of this example, with the version executed at each step, the time each execution took, average time, standard deviation of the execution time, and the lower confidence bound for each version. We use a value of 1 for the k parameter in this example; other values would use the same logic but could make different decisions. The policy starts by executing A, as it is the first version for which we do not have any measurements. Next, it executes A again, as we need at least two executions to calculate its confidence bounds. With these two executions, which took 815 and 1235 ms, it calculates the average and standard deviation, obtaining a lower bound of 1025 ms. Next, it repeats the same for B, running it twice, obtaining 1079 and 858 ms and calculating a confidence bound of 853 ms. As the second confidence bound indicates B may be faster, B runs again next, taking 875 ms to finish and updating its bound to 805. With the executions of B, the bound of A is also updated as the computation of the bound depends on the total number of executions. As the confidence bound of A is now better than that of B, the policy runs A next. However, this execution makes the bound of A worse than that of B, so B runs next. This process continues, B running again whenever the confidence bound of A is considered high enough that seems like a better candidate.

3.2.4 Gradient Bandit

The Gradient Bandit (GB) policy [SB18] mixes exploration and exploitation, and makes use of randomization as part of its strategy. Each task version has a preference. When a task is launched, a version is chosen randomly but weighted by its preference. That way, a version with a high preference has higher odds of being chosen and will be used more often than one with a low preference. Versions with a low preference still have a probability, however low, of executing. By never completely excluding any task version, this policy is guaranteed to eventually identify the optimal task version, if run for long enough.

1	Round	1	2	3	4	5	6	7	8	9	10	...	100
2	Version executed	A	A	B	B	A	B	B	B	B	A		B
3	Time (ms)	815	1235	1079	858	965	674	972	912	976	1060		714
4	**A**												
5	Avg. (ms)		1025			1005					1019		999
6	Std. Dev.		297			122					176		140
7	UCB		1025	850	805	860	849	841	834	828	828		876
8	**B**												
9	Avg. (ms)				967		870	896	899	912			897
10	Std. Dev.				156		203	173	150	138			108
11	UCB				853	838	722	780	805	831	828		873

Table 3.2.: Example execution of the UCB policy. Each column shows the data for one round. Line 1 shows the round of the column; Line 2 shows the version executed for that round; Line 3 shows the time the chosen task version took in that round; Lines 5, 6 and 7 show the average execution time, standard deviation, and lower bound ($\overline{x}_i - \sqrt{k\frac{q_i - n_i\overline{x}_i{}^2}{n_i - 1}\frac{ln(t-1)}{n_i}}$) of A up to that point in the execution, the same statistics for B are shown in lines 9, 10, and 11.

More formally, at round t, each version i has an associated preference $H_i(t)$ and probability $\pi_i(t)$ of being selected that round. Initially, all the preferences are 0 and all the probabilities are $\frac{1}{k}$ where k is the number of versions. The selection probabilities are obtained through a soft-max function, also known as a normalized exponential function. A soft-max function is any function which takes a vector v of k values and normalizes it into a vector u of the same length where $\forall x \in u : x \in [0, 1]$ and $\sum u = 1$. That is, the input vector is normalized into a probability distribution. In our case this is done using the Boltzmann function, also known as the Gibbs distribution, hence the selection probability of some task version a is computed according to the arm preferences thusly:

$$P(A_t = a) = \frac{e^{H_a(t)}}{\sum_{i=1}^{k} H_i(t)}$$

At round t, a version is selected according to the probabilities P. Once the learner receives its reward x_t, it computes

$$x_t - \sum_{i=1}^{t} \frac{x_i}{t}$$

which is used to tell if the reward is better or worse than the average reward until the current round. If the reward is better than average, the preference for that task version is increased, and the preferences for the other arms are decreased as their sum must be 1. Likewise, if the reward is worse than average the preference for that version is decreased and the preferences for the other task versions are increased.

The increase, or decrease, depends on an arbitrary rate α, with a greater increase, or decrease, the greater the difference between the reward and the average:

$$H_{t+1}(A_t) = H_t A_t + \alpha(x_t - \overline{x}_t)(1 - \pi_t(A_t)) \tag{3.1}$$

$$H_{t+1}(a) = H_t a - \alpha(x_t - \overline{x}_t)(\pi_t(A_t)), \qquad \forall a \neq A_t \tag{3.2}$$

Listing 8 shows the algorithm for this policy. Lines 2-7 initialize the preferences (H) and probabilities (π) such that initially every version has the same chance of being selected. Line 8 selects one version at random, according to the probabilities of each version. Notice line 9 computes the difference between the new reward and the average as $\overline{X}_t - X_t$ instead of $X_t - \overline{X}_t$ as we want to minimize the execution time instead of maximizing it. Lines 9-16 update the arm preferences. Lastly, lines 17-19 update the probabilities using the new preferences.

```
1  function GB(k, t, a) {
2    if t = 1 {
3      for i ∈ 1, ..., k {
4        H_i = 0
5        π_i = 1/k
6      }
7    }
8    A_t = random_pick(π)
9    p = α · (X̄_i − X_t)
10   for i ∈ 1, ..., k {
11     if i = A_t {
12       H_i = H_i + p · (1 − π_i)
13     }else{
14       H_i = H_i − p · π_i
15     }
16   }
17   for i ∈ 1, ..., t {
18     π_i = e^{H_i} / Σ_{j=1}^{k} e^{H_j}
19   }
20 }
```

Listing 8: GB algorithm. k is the number of arms, t is the current round, α controls the degreee of optimism, $random\_pick(v)$ returns the index of one element of v chosen according to the probabilities given by that vector, A_i is the action taken at turn i, and X_i is the reward obtained at round i.

To better explain how this policy works we depict how the Gradient Bandit handles a scenario as the one described in Section 3.2.2. We use a value of 0.2 for the

1	Round	1	2	3	4	5	6	7	8	9	10	...	100
2	Version executed	B	A	A	B	A	B	B	A	B	A		B
3	Time (ms)	799	909	1013	1004	975	972	958	965	749	877		729
4	**A**												
5	Avg. (ms)			961		966			966		948		995
6	Std. Dev.			74		53			43		54		101
7	Pref.	1	0.99	0.98	0.99	0.99	0.99	0.99	0.99	0.97	0.97		0.61
8	Prob.	50	49.7	49.1	49.5	49.3	49.4	49.5	49.4	48.4	48.7		31.6
9	**B**												
10	Avg. (ms)				902		925	933		896			908
11	Std. Dev.				145		110	92		114			98
12	Pref.	1	1.01	1.02	1.01	1.01	1.01	1.01	1.01	1.03	1.03		1.39
13	Prob.	50	50.3	50.9	50.5	50.7	50.6	50.5	50.6	51.6	51.3		68.4

Table 3.3.: Example execution of the GB policy. Each column shows the data for one round. Line 1 shows the round of the column; Line 2 shows the version executed for that round; Line 3 shows the time the chosen task version took in that round; Lines 5–8 show the average execution time, standard deviation, preference and probability of A up to that point in the execution, the same statistics for B are shown in lines 10–13. Preference and probability computed with $\alpha = 0.2$

α parameter of the policy[2]. The policy starts by randomly choosing between A and B with the same probability for either version. In this example, it executes B first, taking 799 ms. Notice, in line 7 of Table 3.3 that the preferences of both versions remain the same after this first execution. This is because the preferences are increased or decreased based on how far the execution is from the average of all executions so far, including all versions. That is, an execution faster than average will increase the preference of that task version, and an execution slower than the average will decrease the preference. Since at this point in the execution the average time is the same as the time of the first execution, the preferences remain the same. Next, the policy opts to execute A, obtaining an execution time of 909 ms. As that is higher than the average, the preference, and, consequently, probability of A being selected next are reduced, to 0.994 and 49.7%, respectively. The policy continues choosing randomly between A and B, with a slight bias towards B as it has a higher preference, up to round 10. At round 100, shown in the last column of the table, the preference has further shifted towards B, which then has a 68% chance of being executed.

3.2.5 Comparison Between the Policies

We presented the three task version selection policies we use: Mean, Upper Confidence Bound (UCB), and Gradient Bandit (GB). Mean is a very simple policy, which executes each task version a fixed number of times and, from then on, uses the

[2]The speed at which the preferences and selection probabilities change depends on the value of α. Higher values result in faster changes, and an increased risk of converging towards a suboptimal version.

version that performed best on average. Instead of using a predefined number of executions for each version, UCB runs more times versions with higher variance, provided they have a low enough average execution time, improving its knowledge of the versions it is less certain about. GB uses a different approach, executing versions randomly but weighted by their expected performance.

Mean estimates a version's performance as the average of its past executions. UCB makes an optimistic estimation of a task version's performance, speculating it may be better than what the version has shown in the past on average. This is accomplished by computing a confidence bound for each task version, so while the estimates are optimistic this optimism is backed by past executions. These bounds have an arbitrarily large probability of containing the true expected value of that version's execution time. The trade-off is that increasing the chances of a confidence bound containing the true expectected value means increasing the bound, making it less useful in estimating how well the task version performs. Instead of estimating the expected value of each task version, GB has a preference for each task version. This preference is increased or decreased based on how fast or slow each version behaves compared to the average.

Both Mean and UCB always execute the task version expected to be the best, Mean using the average for this decision and UCB using its optimistic estimation. If Mean makes the wrong decision and the executed version performs badly, this is reflected in the increased average of that version, possibly resulting in a different version being executed next. When a task version performs worse than expectd by the optimistic bounds of UCB, that version's bounds are updated and brought closer to the version's true expected value. GB adopts a different approach, making use of randomization when deciding which task version to run. Instead of using the gathered data to refine estimations of how each task version performs, GB uses this data to change its preference for running each task version. When a version performs well, compared to the average version, its preference is increased, and when it underperforms its preference is decreased. As the preference is used to control the probability of a task version being run, versions with good performance will run more often as the execution progresses.

Mean has two distinct phases in its execution. It first explores all task versions and then exploits the gathered knowledge by always running the version with the best average performance. UCB and GB, in contrast, mix exploration with exploitation. While there are no clearly defined phases, in general these policies are more likely to explore the different task versions in the beginning of execution and are more likely to exploit the knowledge later in the execution.

A major difference with Mean is that UCB and GB never fully exclude any task version. The confidence bounds of UCB often allow, depending how they are computed, a task version that has been executed relatively few times to be executed again even if it underperformed in the past. When selecting a version to execute, GB has a non-zero probability of selecting all task versions, even those which underperformed in the past. As no version is ever excluded, in a sufficiently long execution both UCB and GB are guaranteed to identify the best-performing task version. This guarantee does not come without a cost, however, as both UCB and GB incur the risk of running suboptimal versions more times than Mean. Mean has its own disadvantages however, by limiting the number of times a task version is run it risks being stuck with a suboptimal version no matter how long the execution lasts.

3.2.6 Parameter Sensitivity

As mentioned before, each of these policies can have its behaviour influenced through a parameter. For example, Mean can take fewer samples of each version before committing to one choice to reduce the duration of its exploratory phase, or take more samples to increase its chances of committing to the best version. As another example, UCB can use a small confidence level to exploit its knowledge earlier. As a last example, GB can use a higher preference change rate to converge to mostly using a single version sooner.

One important attribute of a policy is how sensitive it is to the values of its parameters. This is particularly important if certain characteristics of the problem such as the type of the probability distributions are unknown. We verify how sensitive the policies are by executing each policy, with varying values for their parameters, on 1000 problems. To isolate the comparison to the parameters, regardless of differences in implementation between the policies or hardware effects, each problem consists of executing 1000 tasks in sequence, in a single thread. There are 10 task versions, and each of them provides a reward according to a gaussian distribution, instead of according to its actual performance. The distributions used by the versions have their mean generated randomly according to a gaussian distribution with a mean of 1 second, and a standard deviation of 100 ms. The standard deviation of the versions is 100 ms as well. Running versions randomly should provide an average reward of 1 second.

Figure 3.4 shows how each policy behaves in that scenario with varying values for their parameters. Each policy is represented by a line. Each point is the average of 1000 runs of the policy using the parameter value shown in the horizontal axis, with the vertical axis showing the difference, in percent, from always using the best

version for all the 1000 runs. For example, a value of 0 for this difference would mean that policy always executed the best version. We can see Mean, shown in red, is stable so long as the value is not too small, although it performs worse than UCB in all cases. GB, shown in green, is more sensitive to changing its parameter, however it still performs well if the value used is not too low, which makes sense as a value of zero means GB never changes its preferences and always gives every version the same chance of being used. Lastly, we can see the policy which performs the best in this theoretical scenario is UCB, in blue. This policy works well without finetuning its parameter value so long as the confidence level used is not too low.

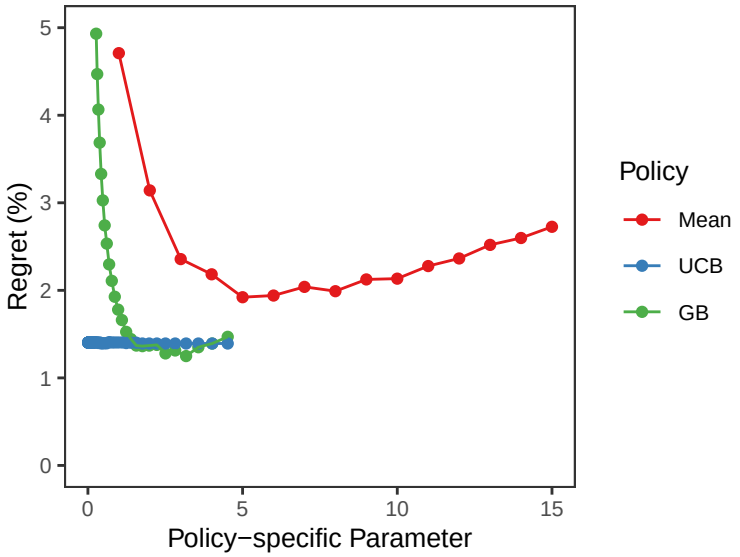

Figure 3.4.: Effect of changing the parameter values in the Mean, UCB and GB policies, shown by lines colored in blue for Mean, red for GB, and green for UCB. Each point is the average of 1000 executions, with 10 versions each. The vertical axis shows how much, in percent, the execution time was worse on average after 1000 runs with 1000 being chosen every time, when compared with always using the best version. The horizontal axis shows the value of the policy-specific parameter.

Parameter sensitivity is an important characteristic since typically the optimal parameter values for some application on a specific platform are unknown a priori. If the policy is too dependent on the values of its parameters, it forces another tuning parameter on the user, and which is contrary to our objective of alleviating the burden of tuning the application.

3.3 Runtime and Compiler Implementation

To be able to experimentally verify the effects of our proposed approach we extended the LLVM [LA04] compiler (Clang) and its OpenMP runtime. We opted for LLVM over GCC as its more modern codebase provides better extensibility. The

GCC project is much older, and despite supporting a larger range of architectures its legacy code makes LLVM a more popular choice for compiler research. Nevertheless, the LLVM compiler supports a wide range of architectures and operating systems, however we only tested our extensions in x86 with GNU/Linux.

Section 3.3.1 describes the additions we make to the OpenMP API. Section 3.3.2 describes our additions to the compiler. Section 3.3.3 presents the changes to the OpenMP runtime.

3.3.1 OpenMP Extension

To support our extension we modify the OpenMP task pragma. In OpenMP a task can be launched with the following syntax:

```
1 {
2   #pragma omp task
3   a = f(n);
4 }
```

When compiled this code is translated into something akin to:

```
1 {
2   task = __kmpc_omp_task_alloc(&a, &n, .omp_task_entry.1)
3   __kmpc_omp_task(task)
4 }
5
6 .omp_task_entry.1(task) {
7   a = f(n);
8 }
```

The data used by the task, in this case the variables "a" and "n", is copied to the task's data structure, in line 2. The task's code is also moved to a new function, shown in lines 6-8. Lastly, the function in line 3 adds the task to the task queue and it may begin executing once its dependencies (none in this example) are satisfied.

We modify this syntax in order to allow the specification of multiple versions for one task. We do this by adding a "define" clause to the pragma. We can then create two versions of the previous task as follows:

```
#pragma omp task define(1)
a = f(n);
#pragma omp task define(1)
a = g(n);
```

The define clause takes one integer argument (1 in the example above), which tells OpenMP which type of task the provided version belongs to. In the example above, one version calls the function "f", and the other calls "g" instead. The compiler translates this code to:

```
.omp_task_entry.1(task) {
  a = f(n);
}

.omp_task_entry.2(task) {
  a = g(n);
}
```

That is, it moves the two versions to separate functions. It does not, however, create a task or copy its data. To create the task we use a second clause, "run":

```
#pragma omp task run(1)
```

This, in turn, copies the required variables and creates the task, being translated by the compiler into something similar to:

```
{
  task = __kmpc_omp_task_alloc(&a, &n, 1)
  __kmpc_omp_task(task)
}
```

The task is allocated similar to before, however instead of the task data containing a pointer to a function it contains the task type, which lets the runtime know it must run some version of the task "1". Furthermore, the variables passed to __kmpc_omp_task_alloc (the local variables "a" an "b" in the example) come from the previous "task defines" instead of the "task run" which merely emits the code.

It is possible to specify the computational cost of a task by calling `void task_cost(long long n)` before launching the task with task run. This func-

tion receives a 64-bit integer, which provides the runtime with the problem size of the next task launched by the calling thread. This information is optional, and when not used the runtime will assume all tasks of that type are of the same size.

The runtime groups together tasks with the same parameters, including the computational cost. However in some applications there are very few tasks which have problems of the same size. In that case it is possible to group together tasks with different, but similar, computational costs. For instance, one could group very small tasks all in the same group, under the assumption that a task version which performs well with one of them will also perform similarly with the others. To group together these tasks with different computational costs one can use `void kmp_task_reltime(long long cost, long long divBy)`, which will place the next launched task in the group of tasks with a computational cost of *cost*. However, since it is meant to group together similar but not exactly equal costs, to keep the measured execution times coherent, the execution time of that task will be divided by *divBy*.

The functions `void task_node1(void* p)` and `task_node2(void* p)` allow the specification of the NUMA node(s) where the data accessed by the next task launched by a thread is located. This can be used when the user expects the best version depend on data locality. That is, the version used when the data is local is not necessarily the same as the version used when the data is not local. Passing this information to the runtime is optional.

The function `void kmp_task_stats_set(kmp_uint64 t)` replaces the execution time for the currently-running task with the 64-bit unsigned integer *t*, which specifies a time in nanoseconds. This function can be used to include, or exclude, the time used by subtasks. For instance, if most of the time used by a task is on its subtasks, it makes sense to include the time spent in those. If, however, there are two versions of the same task type and they create the same subtasks, one may prefer to exclude the time used by the subtasks in order to reduce the noise passed to the runtime as it attempts to find which of the two versions is faster. Since this function replaces the value of the execution time, it requires the user to count the elapsed time. Alternatively, the function `void kmp_task_stats_delta(void* statsp, kmp_uint64 t)` can be used. This function adds the difference between the timestamp *t* and the current time to the recorded execution time. The *statsp* pointer defines the task whose time is affected, and in order to alter the currently-running task, this value must be the one returned by the function `void * kmp_task_stats_get()`. The timestamp can be obtained by calling the function `kmp_uint64 kmp_task_stats_time()`. Another way to change the recorded time of some task is through the function `void kmp_task_stats_add(void* statsp, kmp_uint64 t)`. This function adds the value *t* to the recorded exe-

cution for the specified task. Similarly, but to subtract instead of add, the function `void kmp_task_stats_sub(void* statsp, kmp_uint64 t)` can be used.

The function `void kmp_task_impl_stat_print_csv(int id)` writes to the file *stats.csv* a table containing some statistical data on the tasks executed. This table contains the number of times it was executed and average execution time of every task version of type *id*. This data is further separated by data location and problem size.

The function `void kmp_task_trace_write()` writes a trace of all executed tasks to the file *trace.csv*, with one line for each task. As the trace adds some overhead, before it can be used it must be enabled when compiling the runtime. The *start* and *end* columns show the time at which the task started, and ended, execution, respectively. The *thread* column has the thread identifier of the thread which finished the task execution. The *socket* column contains the number of the CPU socket which finished the task execution. The *address* column holds the memory address where the task version code for the task of that line is located. The *time* column holds the execution time of the task, in nanoseconds. The *cost* column contains the cost of the task, defined by the developer. The *node1* and *node2* columns hold the node numbers where the data used by the task was located, defined by the developer.

The function `void kmp_task_trace_write()` writes a trace of all executed tasks to the file *trace.csv*, with one line for each task. As the trace adds some overhead, before it can be used it must be enabled when compiling the runtime. The trace contains the following columns:

start Time the task started execution

end Time when the task finished execution. Note a task may be preempted, for example if it creates a subtask.

thread Thread identifier of the thread that finished the task execution.

address Memory address where the task version code for this task is located.

time Execution time of the task, in nanoseconds.

cost Cost of the task, defined by the developer.

socket CPU socket which finished the task execution.

node1 Data location of data used by the task, defined by the developer.

node2 Data location of data used by the task, defined by the developer.

3.3.2 Compiler

Our additions to the Clang (LLVM) compiler consist of modifying the code generation of the task pragma. When the compiler finds a task pragma, it outlines the code of the task, moving it out of the calling function and into a new function. If the task code uses no, or only global, variables, the outlining would be done at this point. However, the task code often references local variables, which are located in the executing thread's stack. Furthermore, the code may also use initialized private variables. As the thread executing the task may not have access to these values, as they may be located in the stack of the thread who created the task, these variables are copied to the memory area of the new task, and possibly initialized with the value the variable had at the time of the thread's creation. After the values are copied, the task is queued for execution. It may begin executing immediately, or later, for instance if it has still unmet dependencies. Even if the task has no unmet dependencies, however, there is no guarantee the it starts immediately. The compiler does not (and cannot) copy the data nor does it enqueue the task, what it does is emit code that will, during execution, instruct the runtime to perform these operations. Our two new task pragma clauses, *define* and *run*, split in two the functionality of the task pragma.

The *define* clause does the outlining, moving that task version's code to a separate function. It also copies the required variables. The first change is in how the task data is allocated. The task data has to be allocated by the *define* clause, since it is used to store data generated by each task version. However, only the first *define* allocates this structure, the other *define* clauses add to the same structure as they all are part of the same task. With a single version, the data used by that version is copied to an array inside the task's data. With multiple versions, there is a copy for each version. Likewise, with a single version the size of the data also must be stored, since the array contains other data (even without our additions to OpenMP), but with multiple versions we store the size of the data for each version. Furthermore, instead of the task data containing a pointer to the outlined function, it holds the task type defined by the developer. Tasks with a single version do not have a task type and use the pointer to the outlined version as before. As *task define* does most of the work, *task run* simply takes the generated task data and enqueues the task for execution.

The runtime needs to know which task versions are associated with each task type. More specifically, it must know the addresses of the task versions it can use with each task type. To provide this information the compiler builds a table mapping each task type to all its versions, and stores this table in the executable so it can be accessed by the runtime. This is detailed further in Section 3.3.3.

3.3.3 Runtime

Aside from our modifications to the compiler we modify the LLVM OpenMP runtime to enable it to handle multi-versioned tasks. This section describes those modifications and explains why they are necessary.

Each task version has some data associated with it. Namely, its memory address and the offset used to locate that particular version's data inside the task data, which also contains the data of the other task versions. At the beginning of its initialization, the runtime uses the data provided by the compiler to build a table mapping each task type to the address and offset of each of its versions. During the runtime initialization, we check which task selection policy must be used. This is done through the environment variable *OMP_SELECT*, which defaults to the Eager Mean policy. Possible values are "mean", uses the Mean policy; "ucb", which uses the Upper Confidence Bound policy; and "gb", which uses the Gradient Bandit policy. Furthermore, the parameter of the policies can be chosen with the environment variable *OMP_MEAN_REPS*, for the Mean policy, *OMP_UCB*, for the UCB policy, and *OMP_GB_ALPHA*, for the GB policy.

All required changes in task creation are handled by the compiler, or, more specifically, by the multi-version-aware code generated by the compiler. However, the runtime still needs to handle the changes to task execution. With the single-versioned runtime, when a task is about to be executed the runtime only needs to call the pointer in the task data to the outlined function. Evidently, that is not possible in the multi-versioned runtime. In that case the runtime checks this same pointer. If it is at or after the beginning of the text segment, which is at the address 0x400000 in Linux [Mat+13], by default, execution continues as in the single-versioned runtime. If it is before that address, however, this value is not a function pointer but the identifier of the task type, which means it is a multi-versioned task execution and the pointer will be provided by the task selection policy.

The task selection policy uses statistical data from past executions of that task to decide which version to use. The best version may be different depending on problem size or data locality. Thus statistical data of a task type is separated not just for each task version but for each pair of problem size and data locality as well.

The statistics of each version are stored in a binary tree. There is one tree per task type, and the key used is comprised of the problem size and data locality, as illustrated in Table 3.4. This table shows as an example two versions for the same kernel, and in this case the first version performs better with a problem size of 1024, but with the second version outperforming it when the problem grows to 8192. Data locality is decided based on three factors: two nodes where data accessed by the task is located, and the socket of the processor that will run the task. We only consider whether data is local or not, so the runtime does not take into account whether the data is located in a closer or more distant NUMA node, only whether it is in the node that is executing the task or not. The tree grows as pairs of problem size and data locality are added. However, the tree still allows searches while an item is being added. The downside is a slightly increased cost to insert an item in the tree, but in general that is better than blocking searches in the tree during insertions as those are a minority of operations which would then block all other threads accessing that tree.

Problem size	Data locality	Average	Standard deviation	Count
1024	False	100 ms	10 ms	20
1024	False	150 ms	20 ms	7
8192	False	550 ms	50 ms	10
8192	False	500 ms	70 ms	40

Table 3.4.: Mapping of statistical data to problem size and data locality.

Once the statistical data of past executions is found, the version selection policy chosen by the user is called to decide which version to run. When the version has been selected, the offset for the variables used by that task version is set accordingly and the task begins to execute. When execution finishes, the statistics of that task version with the used problem size and with the used data locality are updated. More precisely, the mean, standard deviation and number of executions are updated at that point. As these values are shared by all threads, the updates are not done concurrently. This means threads may have to wait for a thread to finish updating before they can continue. However, as the computational cost of these updates is very small compared to most tasks, this has little effect on performance.

As mentioned in Section 2.3, the sample average is $\overline{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i$ and the standard deviation of the sample is $s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x}_n)^2}{n-1}}$, where n is the number of times that task version has been executed so far and X_i the number of nanoseconds taken by its i-th execution. As the collected statistics are updated every time a task finishes executing, recalculating the average and standard deviation on every update could

add substantial overhead. These can however be updated without passing through all the points every time by expanding the formulas as follows:

$$\overline{x}_n = \frac{(n-1)\overline{x}_{n-1} + x_n}{n}$$

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x}_n)^2} = \sqrt{\frac{1}{n-1}(\sum_{i=1}^{n} x_i^2 - 2n\overline{x}_n^2 + n\overline{x}_n)}$$

However, in that case, while the mean would be computed correctly, the corrected sum of squares ($\sum x_i^2$), used to calculate the standard deviation, is unstable for large n due to the limitations of floating-point operations when operands are in very different orders of magnitude. To avoid this issue, we use the Welford [Wel62] single-pass method, which does not requires storing every value to compute the standard deviation while also providing stability. This method is based on deriving the corrected sum of squares for $X_i, \forall i \in \{1, \ldots, n\}$ from the corrected sum of squares of $X_i, \forall i \in \{1, \ldots, n-1\}$. By using:

$$X_n - \overline{X}_n = \frac{n-1}{n}(X_n - \overline{X}_{n-1})$$

This shows the corrected sum of squares can be calculated as:

$$\begin{aligned} S_n &= \sum_{i=1}^{n}(X_i - \overline{X}_n)^2 = \sum_{i=1}^{n}(\frac{n-1}{n}(X_n - \overline{X}_{n-1}))^2 \\ &= \sum_{i=1}^{n-1}((x_n - \overline{X}_{n-1}) - \frac{1}{n}(X_n - \overline{X}_{n-1}))^2 + (\frac{n-1}{n})^2(X_n - \overline{X}_{n-1})^2 \\ &= S_{n-1} + \frac{n-1}{n}(X_n - \overline{X}_{n-1})^2 \end{aligned}$$

From that we can calculate the mean using:

$$\mu \approx \overline{X}_n = \overline{X}_{n-1} + \frac{X_n - \overline{X}_{n-1}}{n}$$

And the standard deviation can be calculated with:

$$\begin{aligned} \sigma \approx s_n &= \sqrt{\frac{\sum_{i=1}^{n}(X_i - \overline{X}_n)^2}{n-1}} = \sqrt{\frac{S_n}{n-1}} \\ &= \sqrt{\frac{S_{n-1} + \frac{n-1}{n}(X_n - \overline{X}_{n-1})^2}{n-1}} \end{aligned}$$

Besides the general statistics recorded, each version selection policy updates its own score for the task version after every execution. When finishing an execution of the i-th version at round t, Mean simply uses its mean ($\overline{X}_{i,t}$) as the score. UCB in turn updates the score for that version with $\overline{X}_{i,t} - \sqrt{k\frac{q_{i,t}-n_{i,t}\overline{X}_{i,t}^{\,2}}{n_{i,t}-1}\frac{ln(t-1)}{n_{i,t}}}$. GB has to update not only the score of that version but of the others as well since changing the selection probability of one version alters the probabilities of all other versions as well.

The bandits have a list of the versions, ranking them from highest-scoring to lowest-scoring. These are used when launching a new task to decide which version to run. In the case of Mean and UCB, the version with the highest ranking is selected without looking at the others. In the beginning of the execution the rankings have a very high chance of changing. For most of the execution however the rankings should see few changes. Since changes must be atomic, the code makes use of mutual exclusion to prevent two threads from updating rankings at the same time. However, threads reading the structure can safely do so during an update. This approach reduces the time spent by a thread waiting for another, however the consequence is a thread may choose to execute the highest-ranked version before the ranks are updated. In that case, if the update changes the version which is at the top of the ranking the thread will execute a version which is not the one with the highest score. None of the bandit algorithms used are sensitive to this difference.

To better analyze the behaviour of each selection policy we added the ability to trace the runtime. This trace differs from the standard trace from the LLVM OpenMP runtime in that it includes not only information about which task is running at some time but also about which version of that task. Due to the incurred overhead, to use the tracing capability it must first be enabled when compiling the runtime. To reduce the overhead, the trace is only written to a file at the end of the execution. This of course means the trace may use a high amount of memory if a large number of tasks are launched, but in our experiments this was far below the available memory. If we were to write the trace to a file during execution we would need to write to not a single but one trace file per thread as the writes to a single file cannot be done in parallel. We trace every task execution. Each thread has a different array for storing its traced tasks. This is done so no mutual exclusion is required for the trace. Initially, 32768 positions are allocated for each array, with more being added during execution if necessary. Each traced task uses 416 bits (52 bytes) of memory, as shown in Table 3.5. The meaning of each column of the trace is explained in Section 3.3.1.

64	64	64	64	64	32	32	32
address	start	end	time	cost	socket	node1	node2

Table 3.5.: Size, in bits, of each item in one row generated by the trace.

3.4 Concluding Remarks

This chapter presented our proposal and how it is integrated in OpenMP, both from the compiler and runtime perspectives. This chapter also showed theoretical results for the policies introduced for selecting a task version. That comparison validates our implementation of the policies, as they behave in the way they are expected to in a known scenario. However, by using randomly-generated values intead of real execution times for the versions it does not shows whether the policies perform well in reality. This is addressed in the next chapter, which presents our experimental evaluation which actually executes each task version, without the used of artificial data.

4 Experimental Evaluation

The aim of our experiments is to evaluate the effectiveness of our policies in identifying and using the best version of a task. That is, given a task-parallel application with different versions that can be used to compute each of its tasks, we want to identify, during runtime and with no a priori knowledge, which of these versions should be used so as to maximize the application's performance. The experiments use three policies. The Mean policy is described in Section 3.2.2. This policy executes each version a number of times and then uses the version whose average time is the lowest. The number of times is specified by the user, and defaults to five. The Upper Confidence Bound (UCB) policy is detailed in Section 3.2.3. This policy computes a confidence bound for each task version and runs the version whose bound indicates its true average, as opposed to the average just from its executions, can be the lowest. The Gradient Bandit (GB) policy is explained in Section 3.2.4. This policy executes versions at random, with the odds of each version defined by a gradient function. Versions with lower average execution times have better chances of being executed, however slower versions still have a chance of running.

We use two benchmarks for our experiments: Cholesky and Matrix Multiplication. The first computes the Cholesky decomposition of a dense matrix through the use of four BLAS kernels, namely *spotrf*, *strsm*, *ssyrk*, and *sgemm*. The multiple versions of this benchmark come from two linear algebra libraries, Intel MKL and OpenBLAS. This benchmark is shown in Section 4.2.1. The Matrix Multiplication benchmark consists of a tiled matrix multiplication. It splits the matrices being multiplied into several submatrices. The much smaller submatrices are multiplied using one of several auto-generated versions. The Matrix Multiplication benchmark is shown in Section 4.2.2.

The experimental methodology and the experimental environment, with hardware and software details, are described next, in Section 4.1.

4.1 Methodology

All our experiments are run in Grid5000 [Bal+13] clusters, details of which are given next. Each experiment is repeated 30 times to reduce the effect of noise in

the measurements. In experiments that use more than one CPU socket, the data is distributed among the two NUMA nodes in order to avoid overhelming one of the nodes. In experiments that use a single CPU socket, the data is allocated exclusively on that socket's memory to reduce the access time. OpenMP threads are not allowed to migrate by using *OMP_PROC_BIND*.

4.1.1 Experimental Environment

Two Grid'5000 clusters are used for the experiments presented in this section: dahu and chiclet. The dahu cluster is composed of 32 Dell PowerEdge C6420 nodes, comprised of two Intel Xeon Gold 6130, with 16 cores each, and 192 GiB of memory. The chiclet cluster is composed of 8 Dell PowerEdge R7425 nodes, it has two AMD EPYC 7301, with 16 cores each, and 128 GiB of memory. More details on the hardware used are shown in Table 4.1.

	dahu	chiclet
Processor	2× Intel Xeon Gold 6130	2× AMD EPYC 7301
Microarchitecture	Skylake	Zen
Cores	2 × 16	2 × 16
Hardware Threads	2 × 32	2 × 32
CPU Frequency	2.1 GHz	2.2 GHz
L1 Cache (Instr.)	16 × 32 KiB	16 × 64 KB
L1 Cache (Data)	16 × 32 KiB	16 × 32 KB
L2 Cache	16 × 1 MiB	16 × 512 KiB
L3 Cache	22 MB	64 MiB
Memory	192 GiB	128 GiB

Table 4.1.: Hardware details of the two clusters used for the experiments.

The target platforms run GNU/Linux, kernel version 4.20.7. As we make modifications to the compiler and OpenMP runtime, we must compile these tools to integrate our additions. To do so, we use the GCC compiler version 8.2.1. GCC is only used for compiling LLVM and OpenBLAS. All benchmarks are compiled with our customized compiler, which is built on top of version 6.0.1 of LLVM. In the experiments using Intel MKL, build 20180829 is used. We adopt OpenBLAS version 0.3.7, when used.

4.2 Performance

We use two benchmarks for performance evaluation: Cholesky decomposition, from the DaSH benchmark suite, and a tiled matrix multiplication benchmark. The Cholesky benchmark has three kernels it runs in parallel, and uses two task versions

for each of them. The matrix multiplication benchmark has only one multiversioned task, and uses up to 219 automatically generated versions for it.

4.2.1 Cholesky Decomposition

The Cholesky benchmark is based on the Dense Algebra benchmark of the DaSH benchmark suite [Gaj+14]. It implements a block Cholesky factorization, also known as Cholesky decomposition, decomposing a Hermetian, positive-definite matrix A into the product of a lower triangular matrix L and its conjugate transpose ($A = LL^T$). Cholesky factorization is used for instance for efficiently solving linear equations or Monte-Carlo simulations, which in turn have several applications, such as fluid and cellular structure simulations or to predict failures in engineering problems like aircraft design or oil exploration. The benchmark makes use of the *spotrf*, *strsm*, *ssyrk*, and *sgemm* block-based routines from LAPACK [And+90]. The *sportrf* routine computes the Cholesky factorization of a block; the *strsm* routine solves the matrix equation $XA^T = \alpha B$; the *ssyrk* routine computes the symmetric rank-k operation $C = \alpha A^T A + \beta C$; and the *sgemm* routine performs a matrix-matrix multiplication $C = \alpha A^T B + \beta C$.

Listing 9 shows the code of the benchmark as used in DaSH, omitting OpenMP clauses such as *firstprivate* for clarity. It mixes loop and task parallelism. The function call to *spotrf*, in line 2, is run sequentially, however it is a relatively quick operation compared to the others, with a single instance per iteration of the loop unlike the other operations which are run several times per iteration. The function call to *strsm*, in line 5, is run in parallel by using parallel loop as determined by the pragma in line 3. This loop executes blocks in the same row in parallel. There is an implicit barrier at the end of the parallel loop, ensuring all iterations of the loop have been executed before the program execution can proceed. Consequently, *strsm* does not execute at the same time as the other kernels and never executes blocks of different rows concurrently. After all the *strsm* operations for the row have been executed, there is a parallel region in lines 7–19, as indicated by the pragma in line 7. A single thread executes this region, as shown by the use of the pragma in line 9. While the code in this region is executed by only one thread, other threads can run the tasks it creates. Lines 12–13 launch *ssyrk* tasks, which can execute in parallel with the *sgemm* tasks created in lines 17–18. There is another implicit barrier at the end of the parallel region, before the next iteration of the outermost loop. Consequently, while these tasks can run in parallel they only run in parallel with other tasks created in the same iteration.

We modify this code to use only tasks, removing the parallel loop, as shown in Listing 10. Like in the DaSH benchmark, tasks do not create other tasks. Furthermore,

```
for(j=0; j < NumBlocks; ++j) {
  spotrf( A[j][j], BlockSize);
#pragma omp parallel for
  for(i = j+1; i < NumBlocks; ++i) {
    strsm( A[j][j], A[i][j], BlockSize)
  }
#pragma omp parallel
  {
#pragma omp single
    {
      for(i = 0; i < j; ++i) {
#pragma omp task
        ssyrk( A[j][i], A[j][j], BlockSize);
      }
      for(k = 0; k < j; ++k) {
        for(i = j+1; i < NumBlocks; ++i) {
#pragma omp task
          sgemm(A[i][k], A[j][k], A[i][j], BlockSize);
}}}}}
```

Listing 9: Dense Algebra benchmark from DaSH, which performs a Cholesky decomposition of a dense matrix.

each task has two ways to perform each of the *strsm*, *ssyrk*, and *sgemm* operations: by calling the respective function from either OpenBLAS [Wan+13] or Intel Math Kernel Library (MKL) [Wan+14]. This benchmark uses a single version for *spotrf* since this operation is done sequentially and the overhead of creating a task for it, which would be required to be able to use multiple versions, outweights the benefit of computing it using the most efficient version. Even if there is a large difference in performance between the versions, very few tasks for this kernel would be created compared to the other kernels as the number of *spotrf* executions is linear with regards to the number of blocks, whereas for example the number of times *sgemm* is executed is cubic to that same number.

Three scenarios are used for this benchmark:

4K a single thread, with a 4096 by 4096 matrix as input;

16K 8 threads in a single socket, with an input matrix of size 16384 by 16384;

32K 32 threads in two sockets and an input matrix of 32768 by 32768 cells.

As seen in the benchmark's code in Listing 10, the application groups operations by data locality through the use of blocks, subdividing the problem into several smaller problems. The block sizes we use are those which provide the best performance for each combination of input size and number of threads. The performance with other block sizes and number of threads is shown in Figure 4.1. The first column of this figure shows the results for chiclet, with the same input sizes as scenarios

```
for(j=0; j < NumBlocks; ++j) {
  spotrf( A[j][j], BlockSize);
  for(i = j+1; i < NumBlocks; ++i) {
#pragma omp task declare(1)
    strsm_openblas( A[j][j], A[i][j], BlockSize)
#pragma omp task declare(1)
    strsm_mkl( A[j][j], A[i][j], BlockSize)
#pragma omp task run(1)
  }
#pragma omp taskwait
#pragma omp parallel
  {
#pragma omp single
    {
      for(i = 0; i < j; ++i) {
#pragma omp task declare(2)
        ssyrk_openblas( A[j][i], A[j][j], BlockSize);
#pragma omp task declare(2)
        ssyrk_mkl( A[j][i], A[j][j], BlockSize);
#pragma omp task run(2)
      }
      for(k = 0; k < j; ++k) {
        for(i = j+1; i < NumBlocks; ++i) {
#pragma omp task declare(3)
          sgemm_openblas(A[i][k], A[j][k], A[i][j], BlockSize);
#pragma omp task declare(3)
          sgemm_mkl(A[i][k], A[j][k], A[i][j], BlockSize);
#pragma omp task run(3)
}}}}}
```

Listing 10: Our modifications to the benchmark to make it use tasks instead of parallel loops and two versions of each LAPACK operation, one from Intel MKL and the other from OpenBLAS.

4K, *16K* and *32K*, for the first, second and third rows, respectively. The horizontal axis shows the block size used, and the vertical axis shows the average execution time of the benchmark, in seconds. Different number of threads have their values represented by red circles, for 1 thread, blue triangles, for 8 threads, and green squares, for 32 threads. As shown in the figure, the block size can greatly affect the number of cache hits and misses, and with a too small block size there is significant overhead due to the large number of submatrices, while a too large block size limits the number of operations that can be performed in parallel. We can see that despite the differences between the two architectures and differences in execution time, the two clusters behave similarly: the 4096 by 4096 matrix has its lowest execution time with a block size of 1024 when using one thread, although that size prevents parallelism and a smaller block is better with multiple threads; with 8 threads, the 16384 by 16384 matrix performs best with a block size of 256; and with 32 threads the 32768 by 32768 matrix also performs best with a block size of 256. The number of tasks created by the benchmark when using these block sizes is shown in Table 4.2, varying from 16 for the 4096 matrix to more than 350 thousand with the largest matrix.

Scenario	4K	16K	32K
sgemm	6	2016	8128
ssyrk	6	2016	8128
strsm	4	41664	341376

Table 4.2.: Number of times each of the three kernels is executed depending on the input and block sizes.

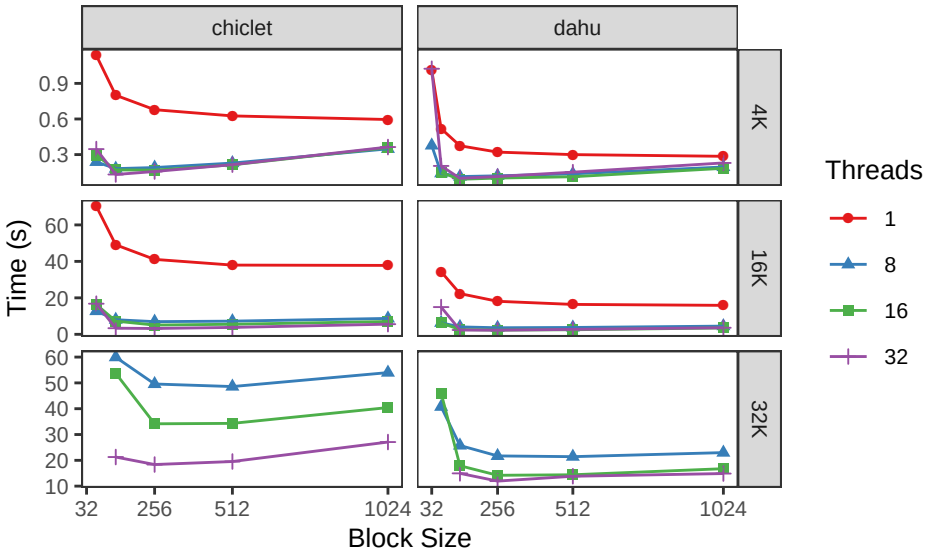

Figure 4.1.: Execution times for the Cholesky benchmark by splitting the matrices using different block sizes.

Each of the matrix operations (*strsm*, *ssyrk* and *sgemm*) has two versions, one that uses the OpenBLAS implementation of the respective operation and the other that uses the Intel MKL implementation. The average performance of using only one of these versions through the whole execution of the benchmark is shown in Figure 4.2, using the best block size for each input size. The first column of this figure shows the results on chiclet, with two AMD processors. The second column shows the results on dahu, with two Intel processors. The first row shows the results for the *4K* scenario, the second for *16K*, and the last column for *32K*. We can see in all cases the OpenBLAS versions outperforms the Intel MKL versions in both platforms.

Our aim, however, is to automatically detect the best version to use during the execution of the application. For that end, we run the Cholesky benchmark using each of the Mean, UCB, and GB selection policies, with parameter values of 30, 16 and 0.8, respectively, and two versions to choose from at runtime for each of the kernels. Figure 4.3 shows the results of these experiments for the three scenarios when using 1, 8 and 32 threads. The horizontal axis shows the selection policy used, Mean, UCB, or GB. The vertical axis shows the average execution time, in seconds. The scale of the vertical axis changes between rows due to their range.

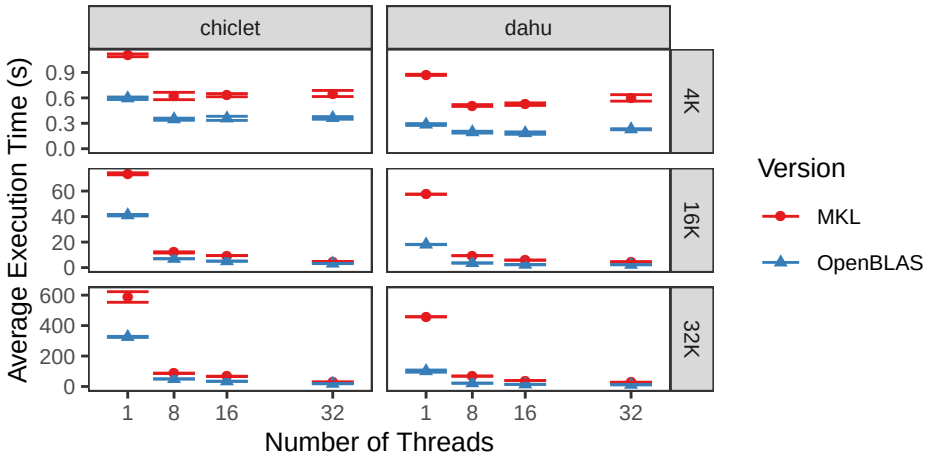

Figure 4.2.: Average execution times for the Cholesky benchmark when using either only the Intel MKL version or only the OpenBLAS version of the kernels.

Lastly, each point correspond to the average execution time of the benchmark, from 30 executions for each combination of input size, platform and policy.

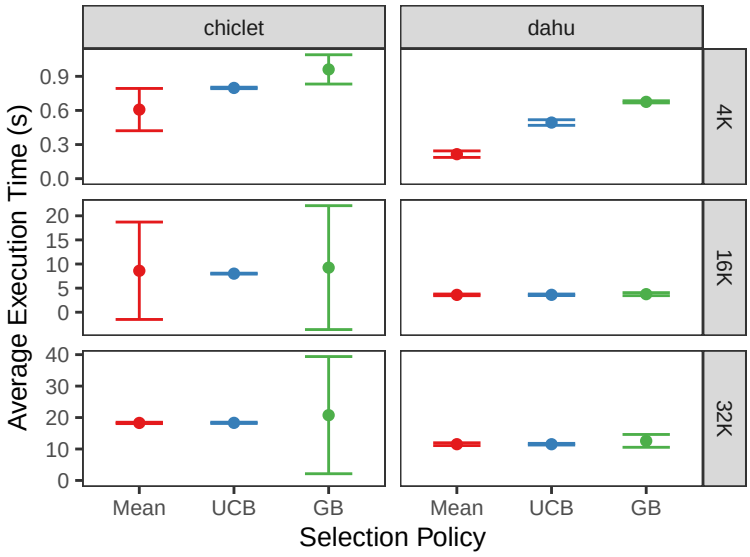

Figure 4.3.: Experimental results for the Cholesky benchmark using each of the GB, Mean and UCB policies, with different input sizes and on two platforms.

While the execution time varies greatly between the two platforms, with dahu executing in less than half the time chiclet does in the 16K scenario in the middle row, the policies behave similarly except with the small matrix in the first row. With the small matrix, the Mean policy performed better than the UCB and GB policies in both platforms, although it showed higher variance in some cases. With the two larger matrices, however, while the GB policy is still outperformed by the two others, the UCB and Mean policies behave very similarly. On dahu and under scenario *32K* the UCB and Mean policies have an average execution time of 11.5

seconds and a t-test cannot identify a difference between the use of UCB and Mean for that problem size.

Previously, in Figure 4.2, we showed the average execution time of the benchmark when only using the best of the two kernel versions (OpenBLAS). Table 4.3 summarizes the executions when using the policies and when using a single kernel. We can see the Mean policy successfully identifies the most efficient versions of the kernels, obtaining similar performance to OpenBLAS in all but one case (*16K* on dahu, where it takes 25% longer to finish). The UCB policy presents similar results, also identifying the correct version in most cases, although with worse performance than the Mean policy in the small *4K* scenario. Lastly, the GB policy seems to have identified the correct version as well, but needed to execute the other version several times and performed worse than the other two policies.

	Mean	UCB	GB	MKL	OpenBLAS
chiclet					
4K	0.6	0.8	1.0	1.1	0.6
16K	8.6	8.0	9.3	11.9	6.9
32K	18.3	18.3	20.8	29.3	18.4
dahu					
4K	0.2	0.5	0.7	0.9	0.3
16K	3.6	3.6	3.8	9.3	3.6
32K	11.5	11.5	12.6	27.6	11.6

Table 4.3.: Average execution times of 30 executions of the Cholesky experiments with 2 versions. The two rightmost columns use a single version for the whole execution.

We decided to trace the execution of the policies to be able to follow their behaviour through the whole execution instead of only knowing the total execution time. To reduce variability, we traced the execution 30 times and looked into the trace with the median execution time. The trace uses a single thread and a 16384 by 16384 input matrix. Figure 4.4 shows the average execution time of each kernel over time during the benchmark's execution on dahu. The horizontal axis shows for how long the application has been running. The vertical axis shows the average execution time, in milliseconds, of that kernel. The GB, Mean, and UCB policies are shown in red, blue, and purple, respectively. The results for when only using the most efficient kernel versions, those of OpenBLAS, are shown in green. The leftmost, center, and rightmost columns show the *sgemm*, *ssyrk*, and *strsm* kernels, respectively. In all cases we can see the UCB policy quickly approached the average of the OpenBLAS version, indicating it quickly found the fastest version. The Mean policy also always identifies the fastest version, however in the middle column with the *ssyrk* kernel we can see that took longer than the UCB policy. Still in the middle column, we can see GB failed to identify the best version of this kernel during the execution, while the other policies were close to the performance of only using the

best version of this kernel. With both the other kernels, GB manages to identify the best version. However, the higher average time tells us it executed the suboptimal version many more times than the other policies did. A trace showing when each version was run during the execution is shown in Appendix B.

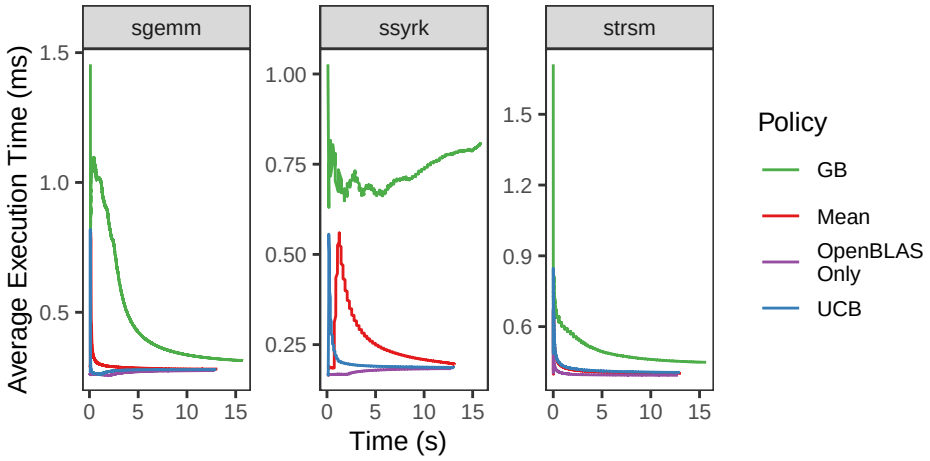

Figure 4.4.: Experimental results on dahu for the Cholesky benchmark using each of the GB, Mean and UCB policies, and for when only using the OpenBLAS version of the kernels.

Figures 4.5, 4.6, and 4.7 show the estimated lower bound of each kernel version with the UCB policy along the first 1000 executions of each of the *sgemm*, *ssyrk*, and *strsm* kernels, respectively. The horizontal axis shows the logical time, one corresponding to the first execution, two to the second, and so forth. The vertical axis shows the time, in milliseconds, taken by each kernel execution. In each figure, the first row corresponds to the OpenBLAS version and the second row to the Intel MKL version. Individual kernel executions have their execution times shown in blue. The red line shows the average time of that kernel version, and the green line the lower confidence bound for the kernel. Each column shows how the lower confidence bound is affected by changing its parameter, with a value of 1, 16, and 32, for the first, second and third columns, respectively. Each column corresponds to an independent execution of the benchmark, so the blue points are not exactly in the same position between columns.

In the first row of Figure 4.5 we can see that through the execution of the OpenBLAS kernels there is a "saw" pattern of a slow kernel execution followed by several faster kernel executions. This is due to the slow execution being caused by a cache miss, which does not happen in the few subsequent executions of the same kernel with data that is adjacent in memory. Because of this, the execution time average starts at a peak due to the initial slow execution and quickly falls. The valley in the beginning of the execution is due to large difference between the timings of the first few executions, which results in computing a large sample standard deviation

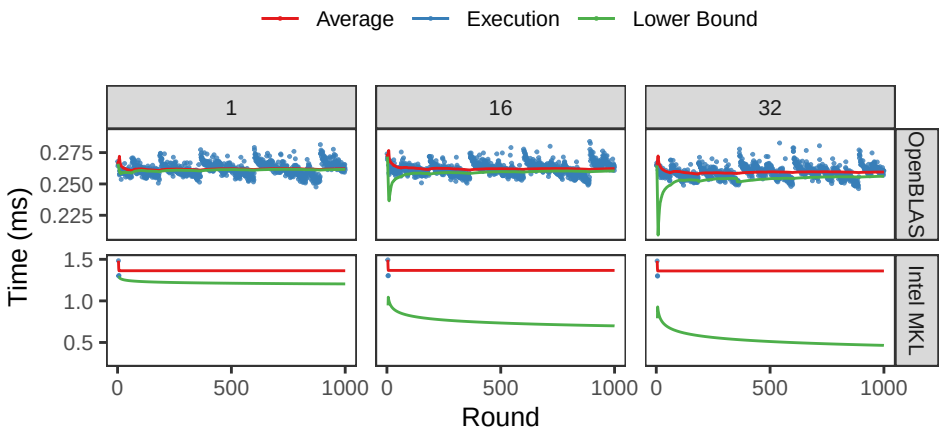

Figure 4.5.: Trace of the execution of the Cholesky benchmark in the 16K scenario on dahu, showing how the *dgemm* kernel behaves through the first 1000 times this kernel is executed in the benchmark.

at this point of the execution (where still few measurements are available), much larger than the true standard deviation of the kernel version. The large distance between the average and the lower bound in the beginning of the execution is to account for the possibility that, just like there was a point much slower than the others, there could be a point much faster. That would mean the true average, too, could be much lower than the average computed at that moment in the execution. However, as subsequent kernel executions have more similar execution times the sample standard deviation quickly decreases and so does the distance between the average and the lower bound. We can see the confidence bound starts large, and approaches the average as more measurements are taken. **The second row**, corresponding to the MKL kernel, however, shows this distance increasing, not decreasing, as the execution advances. This is because the confidence bound also accounts for how often the version has been executed, slightly lowering the bound if the version has been executed relatively few times, which is the case as evidenced by the few number of kernel executions (blue points) in the second row. Lastly, this figure shows the effect of the parameter on the lower bound. With a low parameter value, as shown in the first column, the lower bound becomes closer to the average. Increasing its value, as evidenced in the columns to the right, increases the degree of optimism of the algorithm, increasing the distance from the average.

Contrary to the *sgemm* kernel, the *ssyrk* kernel, shown in Figure 4.6, starts with faster than average executions, causing its average execution time to be initially underestimated instead of overestimated as is the case of *sgemm*. Another difference from the *sgemm* kernel is that the points do not cluster together in clearly distinct subgroups.

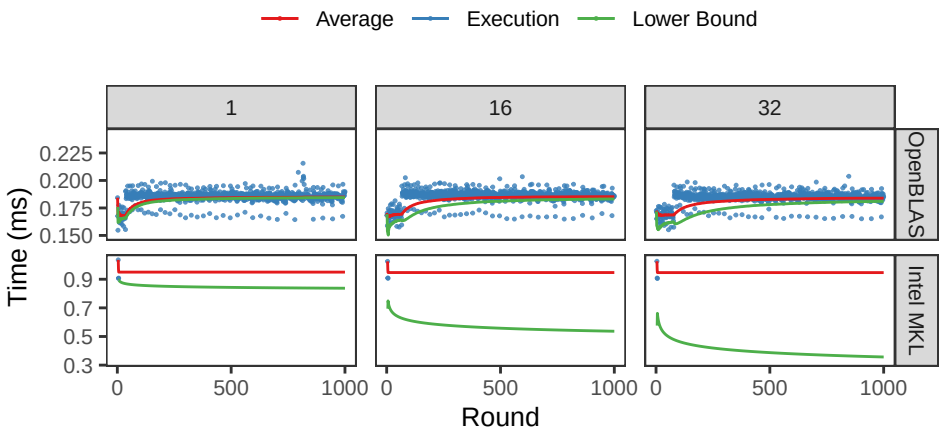

Figure 4.6.: Trace of the execution of the Cholesky benchmark in the 16K scenario on dahu, showing how the *ssyrk* kernel behaves through the first 1000 times this kernel is executed in the benchmark.

Figure 4.7 again presents no clustering except along the average. However, unlike the two previous kernels, which only run the MKL version in the very beginning of the execution, with the *strsm* kernel the MKL version is run a few times during the execution. We can see that while the average execution time is still higher than that of the OpenBLAS version the lower bound is still sufficiently low to sometimes fall below the OpenBLAS lower bound.

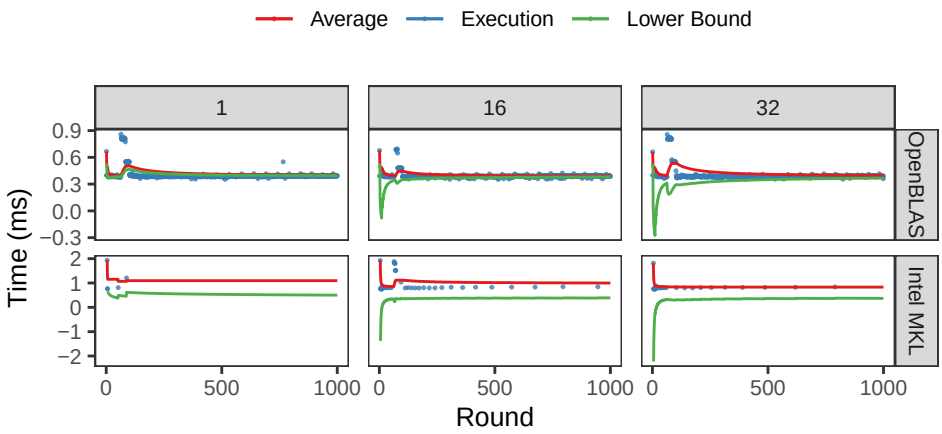

Figure 4.7.: Trace of the execution of the Cholesky benchmark in the 16K scenario on dahu, showing how the *strsm* kernel behaves through the first 1000 times this kernel is executed in the benchmark.

4.2.2 Matrix Multiplication

The Matrix Multiplication (MMUL) benchmark multiplies two matrices in double precision. That is, given two input matrices A and B, it computes a matrix C where

each element of this resulting matrix is the dot product between a row of A and a column of B:

$$\begin{matrix} & & \overrightarrow{b_1} & \overrightarrow{b_2} \\ \overrightarrow{a_1} & \begin{bmatrix} A_{11} & A_{12} \end{bmatrix} & \begin{bmatrix} B_{11} & B_{12} \end{bmatrix} & = \begin{bmatrix} \overrightarrow{a_1} \cdot \overrightarrow{b_1} & \overrightarrow{a_1} \cdot \overrightarrow{b_2} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \end{bmatrix} \\ \overrightarrow{a_2} & \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} & \begin{bmatrix} B_{21} & B_{22} \end{bmatrix} & \begin{bmatrix} \overrightarrow{a_2} \cdot \overrightarrow{b_1} & \overrightarrow{a_2} \cdot \overrightarrow{b_2} \end{bmatrix} & \begin{bmatrix} C_{21} & C_{22} \end{bmatrix} \end{matrix}$$

where A_{ij}, B_{ij}, and C_{ij} are submatrices, $\overrightarrow{a_i}$ is the i-th row of A, and $\overrightarrow{b_i}$ is the i-th row of B. Which is to say the multiplication can be computed through the products and sums of the submatrices. Furthermore, we can see the resulting submatrices $C_{i,j}$ above do not depend on one another and consequently can be computed in parallel. Moreover, each of the resulting submatrices performs two matrix multiplications. For instance, C_{11} is the result of $A_{11}B_{11}$ plus $A_{12}B_{21}$, and can also be further divided into smaller submatrices.

Listing 11 shows the code for the main function of the MMUL benchmark. It begins by verifying if the matrices are sufficiently small to execute sequentially. This is done by comparing the GRAIN parameter against the number of floating-point multiplications required for a sequential computation. The number of multiplications required depends on the dimensions of the matrices and is obtained by multiplying the number of lines in A by the number of columns in B. The result is then multiplied by the number of columns in A (which is necessarily the same as the number of lines in B). This value is computed in line 2. If the value is sufficiently small, line 4 calls the function `solve_sequentially`, which computes the multiplication without further dividing the matrices and will be detailed later. If the number of floating-point operations required is large, however, the matrices are split in lines 6-21, with each submatrix multiplication being launched as a task. Although each matrix is split into four submatrices, 8 tasks are created. This happens because, as mentioned previously, each submatrix containing the result requires two matrix multiplications. So each of these four submatrices creates two multiplication tasks, which are added together. The depend clause of the task pragmas defines a dependency between the two multiplications of each resulting submatrix. This dependency exists because the results of these multiplications are added to C, an operation which has to be done by a single thread. Alternatively, the dependency could be removed if the addition used atomic operations, however, these come with a performance cost.

The function which computes the multiplication of two matrices, without further dividing the problem, is shown in Listing 12. It does so by launching a task which, in turn, uses one of the three functions shown in Listings 13 to 15 to perform the multiplication.

```
1  mmul(m, n, p, A, B, C) {
2    size = m * n * p;
3    if(size <= GRAIN) {
4      solve_sequentially(m, n, p, A, B, C);
5    }else{
6  #pragma omp task depend(inout: C(0, 0))
7      mmul(m/2, n/2, p/2, &A(0, 0), &B(0, 0), &C(0, 0));
8  #pragma omp task depend(inout: C(0, n/2))
9      mmul(m/2, n/2, p/2, &A(0, 0), &B(0, 0), &C(0, n/2));
10 #pragma omp task depend(inout: C(0, 0))
11     mmul(m/2, n/2, p/2, &A(0, 0), &B(0, 0), &C(0, 0));
12 #pragma omp task depend(inout: C(0, n/2))
13     mmul(m/2, n/2, p/2, &A(0, 0), &B(0, 0), &C(0, n/2));
14 #pragma omp task depend(inout: C(m/2, 0))
15     mmul(m/2, n/2, p/2, &A(0, 0), &B(0, 0), &C(m/2, 0));
16 #pragma omp task depend(inout: C(m/2, n/2))
17     mmul(m/2, n/2, p/2, &A(0, 0), &B(0, 0), &C(m/2, n/2));
18 #pragma omp task depend(inout: C(m/2, 0))
19     mmul(m/2, n/2, p/2, &A(0, 0), &B(0, 0), &C(m/2, 0));
20 #pragma omp task depend(inout: C(m/2, n/2))
21     mmul(m/2, n/2, p/2, &A(0, 0), &B(0, 0), &C(m/2, n/2));
22 }}
```

Listing 11: Code for the MMUL benchmark.

```
1  solve_sequentially(m, n, p, A, B, C) {
2  #pragma omp task declare(1)
3    solve_sequentially_1(m, n, p, A, B, C);
4  #pragma omp task declare(1)
5    solve_sequentially_2(m, n, p, A, B, C);
6  #pragma omp task declare(1)
7    solve_sequentially_3(m, n, p, A, B, C);
8  #pragma omp task run(1)
9  }
```

Listing 12: Code for the sequential part of the MMUL benchmark which computes the multiplication with either `solve_sequentially_1`, `solve_sequentially_2`, or `solve_sequentially_3`.

Listing 13 multiplies the two matrices in the most straightforward way possible, simply multiplying and adding the respective lines and columns of the two matrices. Line 4 of this code tells the compiler to unroll the innermost loop for a number of iterations. That is, instead of executing the loop iterations one by one these are grouped in groups of size UNROLL, when possible. In the benchmark, different values of UNROLL are used, resulting in multiple versions from this function.

Listings 14 and 15 show two other ways to multiply the two matrices. Like the previous function, these also make use of loop unrolling in the innermost loop. Unlike the previous version, however, these make use of tiling to improve memory accesses. The tile sizes are given by ITILE and JTILE in both versions, with Listing 15 also using KTILE. In both versions, we use different combinations of values for

```
solve_sequentially_1(m, n, p, A, B, C) {
  for(i = 0; i < m; i++) {
    for(j = 0; j < n; j++) {
#pragma unroll(UNROLL)
      for(k = 0; k < p; ++k) {
        C(i, j) = C(i, j) + A(i, k) * B(k, j);
}}}}
```

Listing 13: Code which multiplies the matrices A and B sequentially.

these parameters. Despite the difference in the tiling, which changes the order the cells of the matrices are accessed, the three functions perform the same number of floating-point operations.

```
solve_sequentially_2(m, n, p, A, B, C) {
  for(ii = 0; ii < m; ii += ITILE) {
    il = min(ii + ITILE, m);
    for(jj = 0; jj < n; jj += JTILE) {
      jl = min(jj + JTILE, n);
      for(i = ii; i < il; i++) {
        for(j = jj; j < jl; j++) {
#pragma unroll(UNROLL)
          for(k = 0; k < p; k++) {
            C(i, j) = C(i, j) + A(i, k) * B(k, j);
}}}}}}
```

Listing 14: Code for the sequential part of the MMUL benchmark.

```
solve_sequentially_3(m, n, p, A, B, C) {
  for(ii = 0; ii < m; ii += ITILE) {
    il = min(ii + ITILE, m);
    for(jj = 0; jj < n; jj += JTILE) {
      jl = min(jj + JTILE, n);
      for(kk = 0; kk < p; kk += KTILE) {
        kl = min(kk+KTILE, p);
        for(i = ii; i < il; i++) {
          for(j = jj; j < jl; j++) {
#pragma unroll(UNROLL)
            for(k = kk; k < kl; k++) {
              C(i, j) = C(i, j) + A(i, k) * B(k, j);
}}}}}}}
```

Listing 15: Code for the sequential part of the MMUL benchmark.

By using different values for the UNROLL, ITILE, JTILE, and KTILE parameters, we generate multiple versions from these three base functions. We generate one version for each possible combination of the values shown in Table 4.4. Listing 13 has three values for one parameter, so it has only three versions. Listings 14 and 15 have multiple parameters, whose combination results in 24 and 192 versions, respectively. In total, there are 219 versions. Since the runtime needs to be told of every version, the code shown in Listing 12 is in fact a simplification of the actual code. In the simplified version shown, there are function calls to only three

functions, each preceded by a pragma. To make use of the 219 versions, this needs to be replaced with 219 function calls, one for each version, each preceded by the same pragma.

Base version	UNROLL	ITILE	JTILE	KTILE	Number of versions
13	enable, 1, 8				3
14	enable	1, 2, 4, 8, 16, 32	32, 64, 512, 1024		$1 \cdot 6 \cdot 4 = 24$
15	enable, 1, 2, 8	1, 2, 4, 8	1, 8, 32	1, 2, 4, 8	$4 \cdot 4 \cdot 3 \cdot 4 = 192$

Table 4.4.: Parameter for generating 219 different versions of Listings 13, 14, and 15.

As we mentioned, the GRAIN size, used in line 3 of Listing 11, defines when the application stops creating tasks and performs the multiplication sequentially instead. When the benchmark splits the matrices it always splits each matrix in four, consequently, if the input is two square matrices of size n the possible values for this parameter correspond to matrices of sizes $\frac{n^3}{8^x}$. For example, if n is 8 the possible values correspond to 8, 4, 2, and 1. This parameter controls not just the size for the sequential execution but also the number of tasks created. In the same example, 0, 8, 32 and 128 tasks are created, respectively, not counting the task created for the sequential execution.

The effect of the GRAIN parameter on performance is shown in Figure 4.8. The first column shows the results for chiclet and the second for dahu. The rows show three different problem sizes, each using two square matrices of sizes 2048, 4096, and 8192 for the first, second and last rows. The matrix sizes are smaller than those of the Cholesky benchmark because Cholesky uses much more optimized BLAS libraries for its matrix multiplications. The vertical axis shows the execution time, in seconds, with a different scale for each input size. The horizontal axis shows the size of the submatrix at which execution is done sequentially. The dots show the average execution time when using only the fastest kernel version, which will be detailed later. The bars show the confidence interval at a confidence level of 99.7%. The colours show the number of threads used, with red for a single thread, blue for 8 threads in a single socket, green for 16 threads in a single socket, and purple for 32 threads in two sockets.

In all cases, we can see the two smallest grain sizes, corresponding to matrices of sizes 8 and 16, are too small, with much worse performance than larger values. We can also see the largest value, 512, is too large for the 4096 and 8192 matrices. For the experiments using multiple versions, we use a grain of 64 for the 2048 and 4096 input matrices, and 128 for the 8192 matrix. We use 128 for the larger matrix as that

grain size performs slightly better than 64 for that input sizes, while 64 performs slightly better for the smaller matrices.

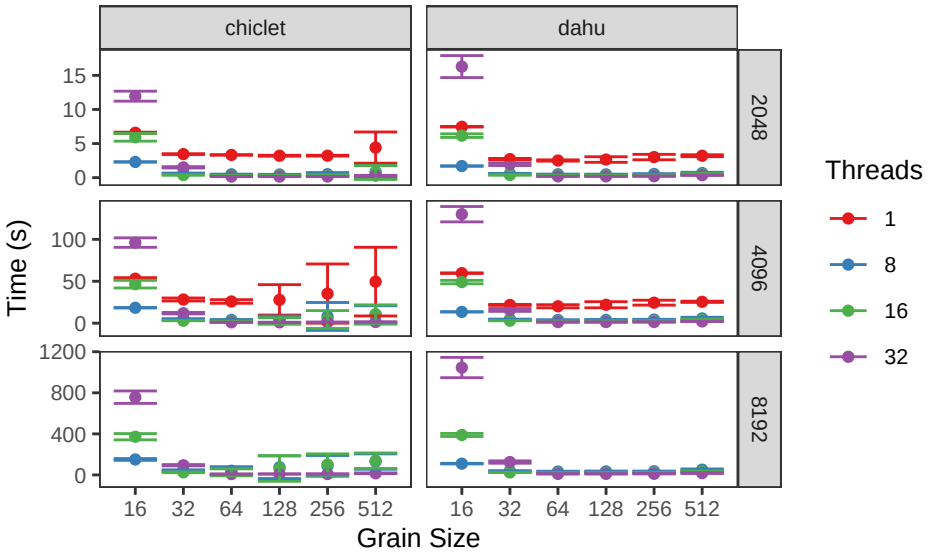

Figure 4.8.: Effect of the grain size on execution time.

Our multi-versioned experiments use 219 versions generated by varying the parameters of the base functions discussed previously. Figure 4.9 shows how each of these versions performs when only that version is used for the whole execution. In the figure, each point is the average execution time. The first column shows the 2048 input, using one thread. The second column shows the 4096 input, using 16 threads in a single socket. The vertical bars represent the confidence interval of these executions at a 99.7% confidence level. The vertical axis shows the execution time, in seconds. The horizontal axis shows the different versions, which are in no particular order. Each base function used to generate the versions is shown in red, blue, and green, corresponding to Listings 13, 14, and 15, respectively. We can see the performance of the first and second base functions varies greatly, but only on chiclet. As these base functions present high variability even when using a single thread, the reason for the variance should not be the task scheduling. On dahu, the parameter values in these two base function have almost no effect on performance, evidenced by their tighter grouping. Furthermore, on chiclet, these two functions are slow when compared to the third base function. Yet, on dahu, the third function can perform worse or better than the first and second functions, depending on the specific combination of values used for the UNROLL, ITILE, JTILE and KTILE parameters.

Figure 4.10 shows the performance of the policies when used with all the 219 versions. The results for chiclet are shown in the first column and those for dahu in the second. The rows show the input sizes we use, each corresponding to multiplying two square matrices with that number of rows and columns: 2048, with

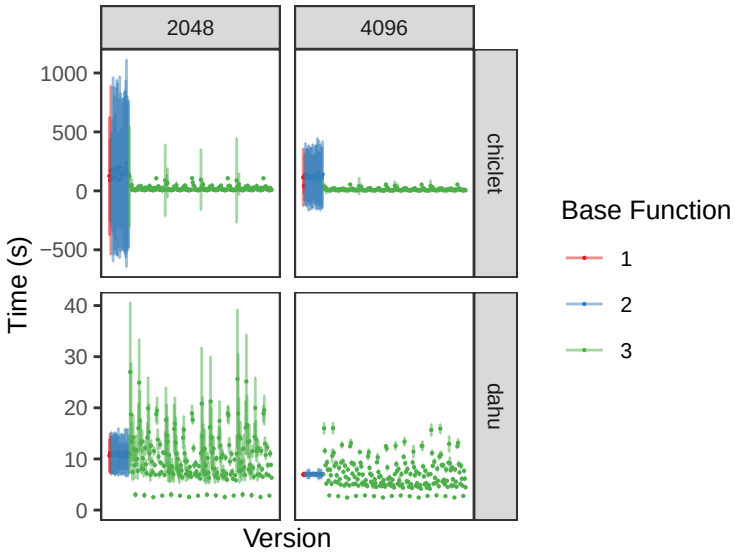

Figure 4.9.: Average execution time of the MMUL benchmark when only using one of the available versions.

one thread, 4096, with 16 threads on a single socket, and 8192, with 32 threads on both sockets. The grain sizes used are 64 for the small matrix and 128 for the others. The vertical axis shows the time, in seconds. The horizontal axis shows the selection policy used, or "Best" for only using the fastest version, according to Figure 4.9, through all the execution. The points show the average time, and the bars show the confidence interval at a confidence level of 99.7%. Lastly, the colour shows different parameter values for each policy. In cases when the execution takes more than 25 seconds the points are shown in Table 4.5.

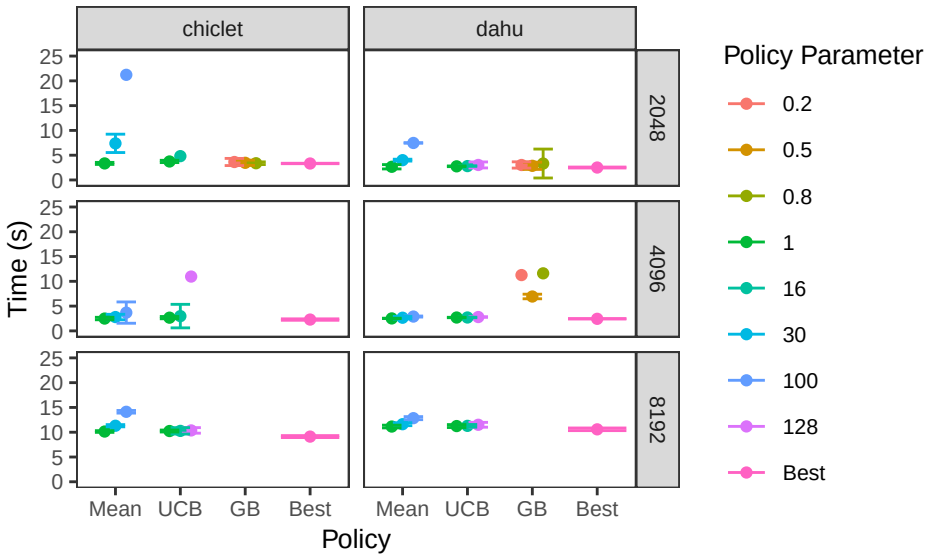

Figure 4.10.: Performance of the MMUL benchmark with the three version selection policies.

	Mean			UCB			GB			Best
	1	30	100	1	16	128	0.2	0.5	0.8	
chiclet										
2048	3.3	7.4	21.2	3.7	4.8	63.7	3.6	3.5	3.4	3.3
4096	2.5	2.8	3.7	2.7	3.0	11.0	172.3	315	297.8	2.3
8192	10.1	11.3	14.1	10.2	10.3	10.4	472.9	434.3	443.4	9.1
dahu										
2048	2.7	4.0	7.5	2.8	2.8	3.0	3.0	2.9	3.3	2.5
4096	2.5	2.7	2.9	2.7	2.7	2.8	11.3	6.9	11.6	2.4
8192	11.1	11.6	12.8	11.2	11.3	11.5	62.5	35.8	59.6	10.6

Table 4.5.: Results of the Matrix Multiplication experiments with 219 versions, shown as the average execution time, in seconds. The first row shows each of the three policies, and Best for the best version previously found. The second row shows the parameter used with the policy.

The first row of Table 4.5 shows the three policies behave similarly on dahu, however with the Mean policy only performing well with a parameter value of 1, which corresponds to a single execution of each version before the policy starts using the one whose average time is the lowest. With the two larger inputs, the GB policy performs much slower than the other two, taking longer than 2 minutes with the 4096 input, and more than 5 minutes with the largest input. Furthermore, we can see the Mean policy again performs best with a parameter of 1.

The Mean policy takes 6.6% longer than always using the best version previously found, when using the best value for its parameter with each of the inputs, which happens to be 1 in all cases. The UCB policy takes on average 10.2% longer than that version, when using 1 for its parameter, and 14.9% when using the canon parameter value of 16. The GB policy performs well in a few cases, however, in most cases it does not manage to obtain a good execution time, being 2000.7% slower on average even when always using the best value for its parameter, as it took several minutes on chiclet.

Looking at these results alone could lead one to think the Mean policy, only executing each version once (i.e. using a parameter value of 1), is sufficient to always identify the best-performing version. By running each version only once, the Mean policy risks missing the best-performing version. This can happen, for instance, if, by chance, that single execution happens to be much slower than its average time. However, if we look back to Figure 4.9, which shows the performance of each version when used alone through all the execution, we can see many of these versions present very similar performance. Since there are many versions which perform similar to the best-performing version, in this particular case the policy has not only a single but several chances to detect one of the best versions.

To test how the policies perform with a set of versions which are more similar we run the same experiment but instead of using all the 219 versions we use only 13 of those. The versions used were some of the best-performing versions, aiming to have versions that have similar performance. Figure 4.11 shows how each of the chosen versions performs when used by itself during the whole execution of the application. In the figure, the vertical axis shows the time, in seconds. The horizontal axis shows each of the 13 versions. As the versions were selected to perform similarly, not to simply maximize performance, the best-performing version found previously is not among them. It is, however, shown in the figure, in blue in the right, and we can see that while the best-performing version showed a slightly lower execution time than the second-fastest version on chiclet this difference is not significative given the number of versions we compare.

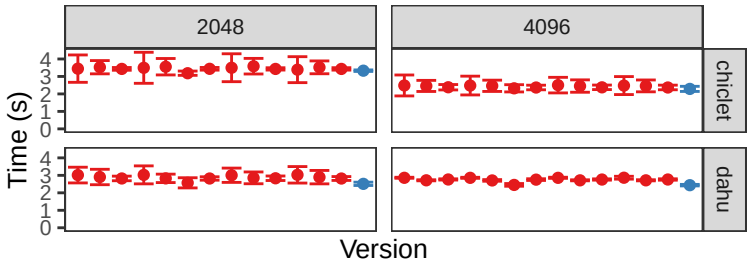

Figure 4.11.: Average execution time of the MMUL benchmark when only using one of the available versions.

The results of the experiments using this smaller set of versions are shown in Figure 4.12. Contrasting with Figure 4.10, there is no case where a policy performs much worse than the best version. This is expected since, in this case, all the versions available to the runtime perform similarly. The values in this figure are shown numerically in Table 4.6.

	Mean			UCB			GB			Best
	1	30	100	1	16	128	0.2	0.5	0.8	
chiclet										
2048	3.2	3.1	3.2	3.2	3.2	3.2	3.1	3.2	3.2	3.3
4096	2.0	2.0	2.0	2.0	2.0	2.0	2.1	2.1	2.0	2.3
8192	8.8	8.8	8.9	8.9	8.8	8.8	12.0	11.9	11.8	9.1
dahu										
2048	2.5	2.6	2.5	2.5	2.7	2.7	2.6	2.5	2.5	2.5
4096	2.8	2.5	2.8	2.8	2.6	2.6	3.0	2.9	2.9	2.4
8192	11.6	10.5	10.4	11.4	10.9	10.6	13.3	11.9	11.9	10.6

Table 4.6.: Results of the Matrix Multiplication experiments with only 13 versions available. The first row shows each of the three policies, and Best for the best version previously found. The second row shows the parameter used with the policy. The values are the average execution time, in seconds.

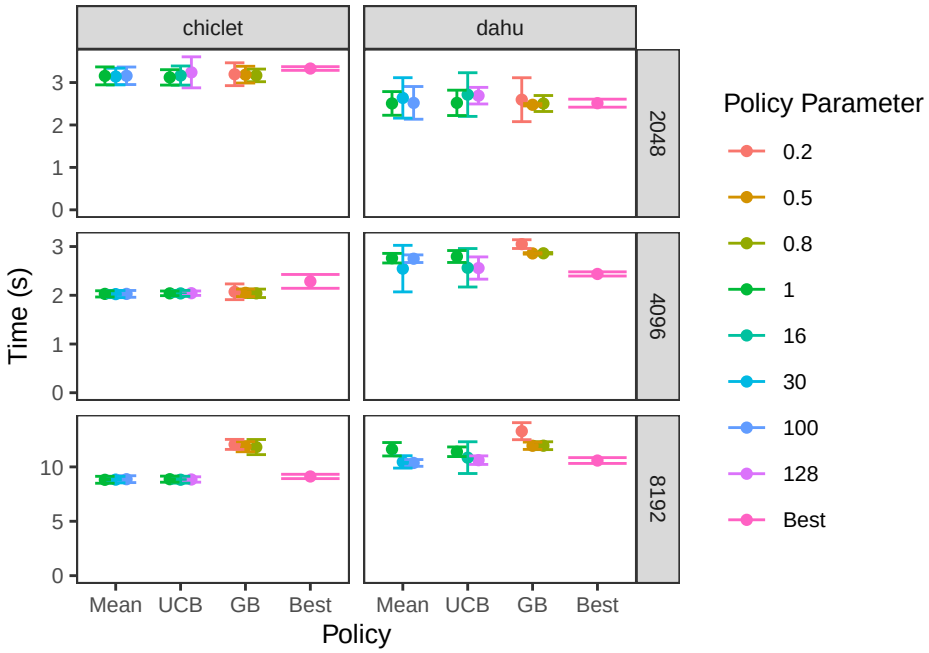

Figure 4.12.: Performance of the MMUL benchmark with the three version selection policies and 13 versions to choose from.

In chiclet, in the first two rows, the three policies perform better on average than the version we previously found to be the best, and both Mean and UCB also outperform it on the last row. This difference is significative at a 99.7% confidence level for all policies and input sizes, except for the GB policy with the largest input, in the third row. This is mainly due to two factors. Firstly, the best version was chosen based on a single input size and number of threads. That could explain the first row, but not the second as the best version was chosen using the same input and number of threads. Secondly, and perhaps more importantly, the best version was chosen based on only 30 executions of the application for each version. While 30 executions can be sufficient for instance to tell how two BLAS libraries or compilers affect some application when this difference is somewhat large, due to the large number of versions and the small differences between their means, this number is not enough to identify the actual best version among hundreds with certainty. The runtime meanwhile has access not only to 30 executions of the whole application but to thousands of task executions using each version.

While the improvement on performance is high on chiclet, being up to 11% faster with the 4096 input, on dahu this improvement was only seen in three cases, GB with a parameter of 0.5 with the 2048 input, and Mean with parameter values of 30 and 100 with the 8192 input. While in these cases there is a significant difference, at a confidence level of 99.7%, when compared with the best version, it is much more modest than in chiclet, being of only 1% on the small input and up to 2% on the largest input.

On dahu and with the 8192 input, in the last row, the Mean policy shows the opposite behaviour it shows in the experiment with the 219 versions. With many versions, the policy performs better when executing each version just once before switching to always using the version whose average time is lowest. With only a few, and very similar, versions, however, the policy performs better by executing each version more times (30 and 100, shown in light blue and blue, respectively). The behaviour of the same input on chiclet also changes, with many versions to choose from it was better to use a single repetition in this case, but with few versions, differently from the case of dahu, the number of repetitions used does not significantly affect the performance.

To explain the difference in the behaviour of the Mean policy when using few or many versions we traced its execution, recording every decision made by the policy and when. As the most interesting case is found with the 8192 input, we use that input and the same number of threads (32) in the trace. To reduce the environmental noise in the measurements we execute the trace 30 times and use the one with the execution time closest to the median. Tracing undoubtly adds some overhead. In the worst case, the median version with tracing enabled took 12% longer to finish than when executing without tracing.

Figure 4.13 shows the traces. The first column shows the behaviour of the Mean policy, and the second shows the UCB policy. The first row shows the behaviour when using only 13 versions, with the second row showing how the policies behave with all the 219 versions. The colours show the different values for the policy parameter. The horizontal axis shows for how long the application has executed so far, in seconds. The vertical axis shows the average execution time of the sequential multiplication tasks up to that point in the execution, in milliseconds. The vertical axis starts on 1, not 0, to better show the small differences between the parameters for each policy. The lines show the average execution time of all the sequential multiplication tasks executed up to that point in the execution, from any of the 32 threads. The dashed lines show the average execution time of the fastest version found so far. This average time may increase if subsequent executions happen to be slower since it is the average computed up to a given point in the execution. Lastly, the vertical lines indicate where each execution finished.

The bottom right of this figure shows the UCB policy, with 219 versions, and three parameters. We can see they behave very similarly and end at roughly the same time, which is coherent with the behaviour it showed before tracing the execution. The top right shows the same policy but with 13 version to choose from. This corresponds to the bottom right of Figure 4.12, which shows the parameter value of 128 outperforms the value of 1, with the value of 16 standing between the two. As the dashed lines show, while the policy quickly settles on a version to use most

of the time, with a parameter of 1 this version is slower than the version found in the other two cases and consequently in this case the policy takes longer to finish execution.

On the bottom left of the figure we have the Mean policy with three values for its parameter, 1 in red, 30 in green and 100 in purple. These three executions correspond to the bottom right of Figure 4.12, which uses 219 versions. The vertical lines show the fastest execution used a single repetition, the second-fastest used 30 and the slowest used 100, as expected. The dashed lines on the left show that regardless of the value of its parameter the policy quickly identified a version to use and that version has similar performance in the three cases as the dashed lines appear one over the other. We can see that with the 100 repetitions the policy takes longer to converge to the time of the best version it found, with no benefit since it does not find a better version to use. The top left of the figure shows how the policy behaves when using only 13 versions. While we can see the use of 30 or 100 repetitions finds similar versions to use, and results in similar execution times, by using 1 repetition the policy fails to identify that version. The drop in the dashed red line close to 5s is due to the policy finding a better version, however, it again failed to identify a version as good as the other two cases.

Figure 4.14 shows the first second of execution the two policies, with the UCB policy using a parameter value of 16 shown in blue and the Mean policy shown in red for a value of one and purple for 100. At the top of this figure, the executions with 13 versions are shown. We can see the UCB policy identified the same best version just slightly after the Mean policy with the best parameter for this case, 100. At the bottom, with all the 219 versions, we can see the UCB policy behaved very close to the Mean policy, but this time to when it used a parameter of 1, which is the best parameter value for this case.

While the Mean policy often performed best, that is only when it used the best number of repetitions for that particular problem. The UCB policy tended to be slightly slower than UCB, however, tended to perform well with a value of 16. The GB policy performed well when used with the 13 versions, although it tended to be outperformed by the other two policies, however, it performed badly with the 219 versions.

Figure 4.15 presents a summary of the previous results for the benchmark. The first column shows the results on chiclet and the second on dahu. The first line shows the results with 219 versions and the second with 13 versions. Instead of absolute values, the vertical axis shows a percentage of the best version. The horizontal axis shows the different input sizes. The colours show the Mean policy with 1 and 100 repetitions and the UCB policy with a parameter value of 16. The figure omits the

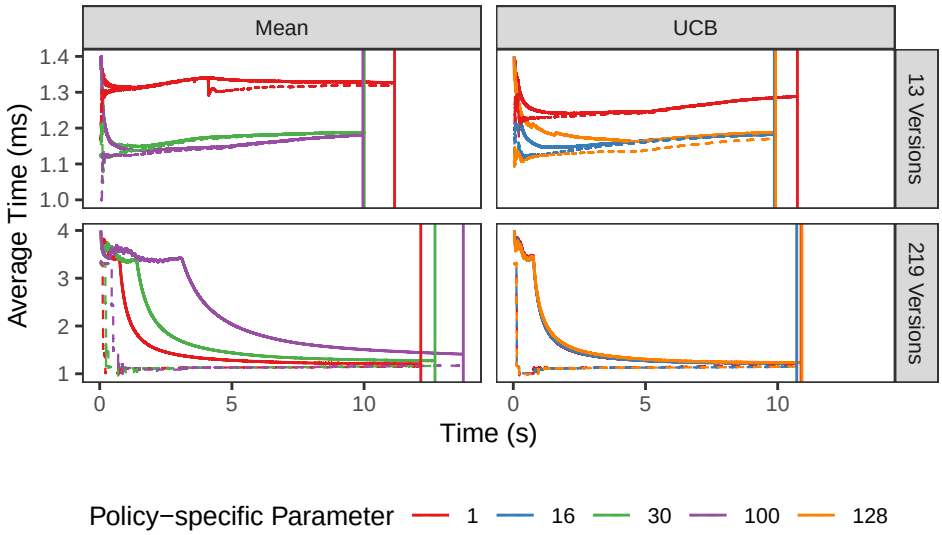

Figure 4.13.: Trace of the Mean and UCB policies, with different parameter values, using two square matrices of size 8192 as input and 32 threads in two sockets.

Mean policy with 100 repetitions on the smallest input when using 219 versions since, as shown previously, it performed very badly and would make it difficult to see the other values.

4.2.3 Concluding Remarks

We show experiments using two benchmarks, a Cholesky computation from the DaSH benchmark suite [Gaj+14], and a tiled Matrix Multiplication. The Cholesky benchmark has two versions for each of its kernels, one version using Intel MKL and the other using OpenBLAS. The Matrix Multiplication benchmark is used for two different experiments, one with 219 versions for its single kernel, and the other with 13 versions.

In the Cholesky benchmark, the Mean and UCB policies successfully identify the best versions to use in most cases. The UCB policy cannot identify the version quickly enough on the smallest input as it does not generate enough tasks. The Mean policy in most cases identified the best version even when running each version only a single time, due to the large difference in execution time between the two versions. The GB version failed to approach the performance obtained by only executing the best version. Even with the largest input, it still tended to execute both versions, resulting in high variance and higher average execution time than the other policies. The Cholesky benchmark is, however, an easy problem since it only has two versions, and those have a high difference in execution time.

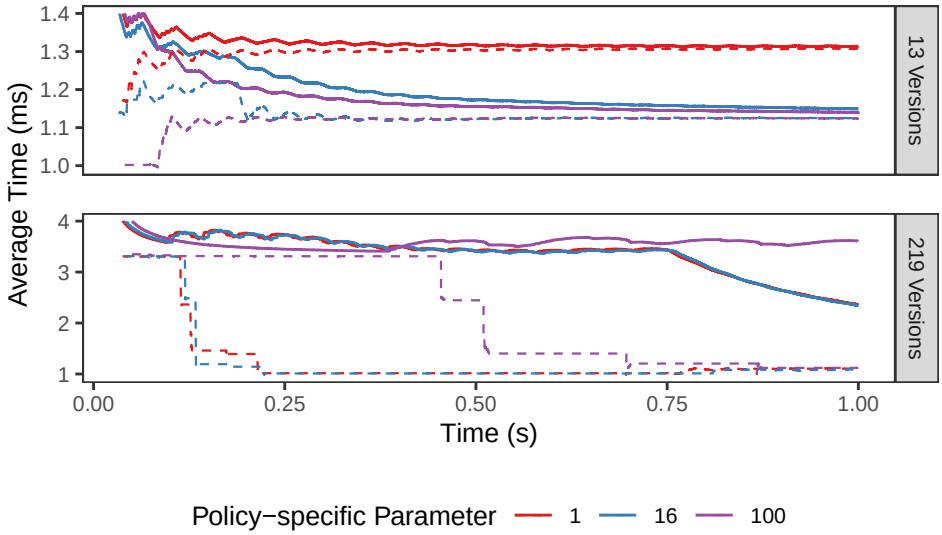

Figure 4.14.: First second of the trace of the Mean and UCB policies using two square matrices of size 8192 as input and 32 threads in two sockets.

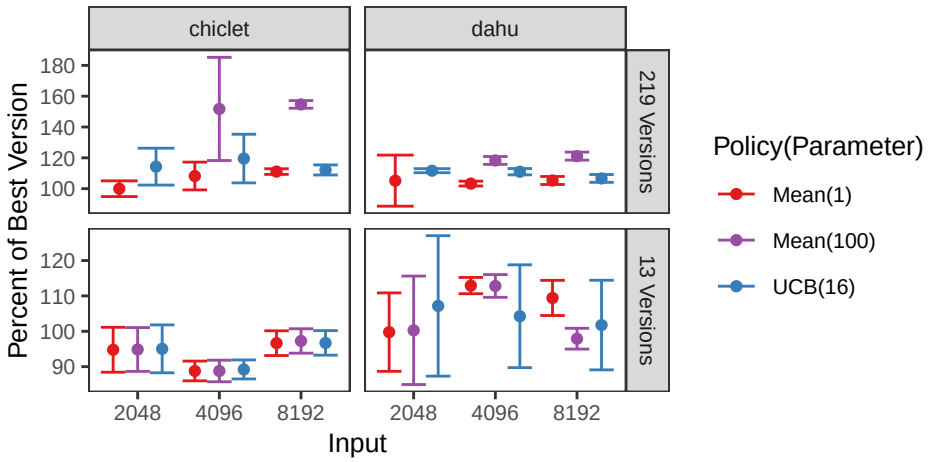

Figure 4.15.: Summary of the Matrix Multiplication results for the Mean and UCB policies.

As the first of the Matrix Multiplication scenarios has several (219) versions, and many of these perform similarly, it favours executing each version only a few times. Many executions are not only superfluous but can decrease performance, and even with the largest input running each version several times results in degraded performance. The second scenario, with fewer (13) task versions available, favours a higher number of executions of each version. While it is easier in the sense that there are fewer versions than the previous scenario, its challenge lies in that the versions presented are very similar to each other. Like with the Cholesky benchmark, the best policy tended to be the Mean policy. However, for the policy to perform well, it requires correctly adjusting the number of times each version must be executed, require more repetitions with the few similar versions and fewer repetitions with the 219 versions. The UCB policy, on the contrary, works well without the need to

change its parameter for either scenario, presenting a safer alternative to the Mean policy, at the price of not being as rewarding as the Mean policy with the right number of repetitions.

5 Related Work

In this chapter, we describe the approaches taken by works which handle the problem of selecting which of multiple task versions to use and how they compare to ours. OmpSs and StarPU are two parallel programming frameworks which support the use of multi-versioned tasks. We describe them in Section 2.2. Aside from their similarities to OpenMP, of particular interest are some of their task schedulers which handle multiple versions of the same task, possibly for different devices. We first describe some of the schedulers used by OmpSs, in Section 5.1. Next, we describe some of the StarPU schedulers in Section 5.2. Lastly, we compare the approaches of OmpSs and StarPU with ours in Section 5.3.

5.1 OmpSs

When using OmpSs the user can choose one of several scheduling policies. Each targets a different scenario, and their characteristics vary from a single queue to several queues depending on the resource type, with different approaches to handle the use of idle resources and the order tasks are executed.

The Breadth-First scheduling policy shares a single queue among all the workers. When dependencies allow, tasks are executed in a first in first out (FIFO) or last in first out (LIFO) fashion, as specified by the user.

The Distributed Breadth-First policy executes tasks in a FIFO or LIFO order, given the tasks have their dependencies met. Each worker adds tasks to and gets tasks from its local queue. When its ready queue is empty, the worker executes the current task's parent, if it is in the ready queue of another worker. If the parent is already being executed or is not ready due to unfulfilled dependencies, the worker steals a task from the next worker's ready queue. Stealing is done from the opposite side of the queue (i.e. the last task is stolen when using FIFO order), to reduce contention between the thief and the victim.

The Work First scheduling policy has one queue per worker. This policy, when using the default values, behaves equivalently to Cilk. New ready tasks begin executing immediately, pushing the current task into the ready queue. By default,

it uses FIFO order for the local queue and LIFO for steals. When the local queue is empty the worker steals the parent task, if possible.

The Socket-Aware policy uses one queue per NUMA node. Each top-level task, that is, the tasks created by the main application and not by other tasks, is assigned to a user-defined NUMA node. Subtasks are assigned to the same NUMA node as their parent. By default, when a node's ready queue is empty its workers steal from the closest nodes, although this behaviour can be altered to allow other nodes, to attempt to steal top-level tasks or to disable task stealing altogether.

The Bottom Level-Aware scheduling policy is made for single-ISA heterogeneous architectures like the ARM big.LITTLE, which have asymmetric cores, some being fast while others are more energy-efficient. This policy uses a different queue per core type, so all the slow cores share one queue and all the fast cores share another queue. Tasks belonging to the longest path in the task dependency graph are assigned to the fast cores. By default, work-stealing is only used by the fast cores. The policy can, however, also use work-stealing for the slow cores.

OmpSs supports CPU and accelerator tasks. A task can have a CPU version and a GPU version, for example, or even multiple CPU and GPU versions. Tasks of the same type must use the same parameters and have the same dependencies. As a task can be launched using different data, for instance, the same operation may be done in a small or a large matrix, OmpSs requires the user to annotate the source code with the size of the data the task is handling. Different input sizes can affect not just how long a version takes to execute, but it is possible the fastest version for a small input is not the same as the fastest version for a larger input. This size information is used in the data structure illustrated in Table 5.1, which depicts two tasks, the first with three versions and the second with two implementations. The timing data collected for each version is partitioned according to the data size, shown in the middle column of this table.

TaskVersionSet	DataSetSize	<VersionId, ExecTime, #Exec>
task1	2 MB	<task1-v1, 30ms, 200> <task1-v2, 18ms, 350> <task1-v3, 25ms, 230>
	3 MB	<task1-v1, 45ms, 80> <task1-v2, 25ms, 300> <task1-v1, 40ms, 120>
task2	5 MB	<task2-v1, 15ms, 40> <task2-v2, 20ms, 3>

Table 5.1.: Data structure used by OmpSs to map version timings to a specific task type and problem size.

The scheduling policies mentioned previously were not made with multiple versions in mind. To handle this feature, OmpSs uses the Versioning scheduling policy [Pla+13]. This policy decides not just which task to run next but which version it should use. Each worker is responsible for a single device, be it small like a CPU core or large like an accelerator. So, for example, a system with 32 cores and 2 GPUs can have 34 workers. Each worker has its own task queue.

There are two distinct phases for each task type: exploration and exploitation. At the beginning of the execution, when the scheduler still has little information about the task versions, ready tasks are distributed to the workers with the version used being chosen in a Round-Robin fashion. Each task is run at least k times, as decided by the user. Every time a task is executed, even after the exploration phase, the policy updates the task's average execution time according to the arithmetic mean. The average execution time is independent for each data size, as shown in Table 5.1. After the initial learning phase, the scheduler starts to make use of the information it gathered thus far. Every combination of task type and input size is independent, so one task type may be still in the exploration phase while another is in the exploitation phase.

Once in the exploitation phase, the scheduler assigns tasks to their earliest executor. That is, tasks are assigned to the worker who should finish the task the soonest, according to the gathered statistics. When all workers are free, this means a task is run using the fastest task version, on the fastest worker for that task. A task can, however, be assigned to a slower worker. This can happen if the fastest worker is busy and is expected to remain busy for a long enough time that the task can be finished sooner by a less efficient, but more available, worker.

5.2 StarPU

Two of StarPU's scheduling policies support multiple implementations [BCI19]: the Deque Model Data Aware policy (DMDA) and Parallel Heterogeneous Earliest Finish Time policy (PHEFT). The DMDA policy follows the HEFT strategy, which is to assign a task to whichever worker can finish it first. This means tasks can be executed by a worker which is not the fastest for that particular task. This can happen in a situation like the one depicted in Figure 5.1, which shows one task with two versions, one for GPU, in red, and another for CPU, in blue, and two workers, one with tasks in its queue and another with an empty queue. In this case, the GPU version is faster than the CPU version. However, DMDA schedules the task using the CPU version as the GPU has enough queued tasks that the CPU, despite being

slower, can finish first. DMDA schedules a task as soon as it is available, not taking priorities into account when making scheduling decisions.

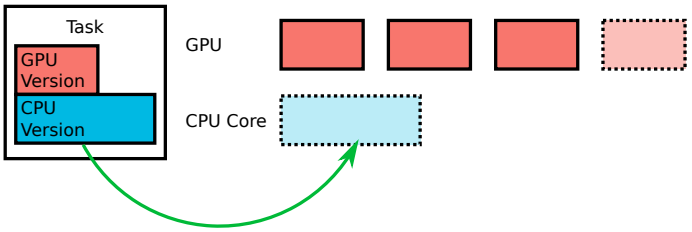

Figure 5.1.: A task with two versions being scheduled in one of two workers.

One limitation of the DMDA policy is it views CPU cores and accelerators at the same level, both are considered a worker. However, for certain types of computation, accelerators, with possibly thousands of cores, tend to be much much faster than a single CPU core. This can make it difficult to make use of the CPU cores since if there are not enough tasks it may be faster to leave all cores idle for much of the computation. One could always create more fine-grained tasks, however, that comes at the price of increased overhead. The PHEFT scheduler, which as the name implies also uses the HEFT strategy, attempts to solve this by having a worker handle a group of cores. For example, a socket with 16 cores can be managed not as 16 workers but as a single worker. This enables the creation of parallel tasks on the CPU, in turn making possible the use of the parallel version of libraries, for instance for the use of the CPU version of a BLAS library.

The aforementioned policies require a performance model to be able to estimate, in advance, how long some task version should take to execute. Like with OmpSs, this estimation uses a structure similar to the one depicted in Table 5.1, with separate measurements for different input sizes. StarPU supports a few different approaches for its performance model. The user can provide the runtime with the expected execution time of a task by using the Common or Per Arch models. For example, the execution time can be predicted using the input size and the number of floating-point operations required for the computation. However, that option is often unrealistic due to the complexity of computer architectures. An alternative is to use a regression model with one of the Regression models. In this case, the user runs the application several times, with varying parameters, so that the runtime can collect the timing information for the different combinations of parameters and build the regression. The regression is built using the least squares method, that is, it attempts to minimize the sum of the squared differences between the predicted and the measured performance for each of the executions. This is illustrated in Figure 5.2, where the line shows the regression, the points show the time measurements, and the red lines show the differences between the two. The least squares method builds its predictor to minimize the sum of the squares of these red lines. The effectiveness of the regression depends of course on the kind of computation being

performed, working well if the performance is regular in regards to the parameters. The performance can vary in a way a simple linear regression cannot always accurately predict. In that case, it is possible to use multiple regressions or non-linear regression, although a non-linear regression is much more computationally expensive and must be computed offline [ATN10].

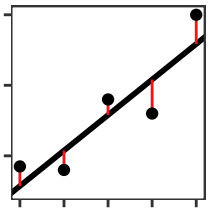

Figure 5.2.: Example of the working of the least squares method.

These performance models require some programming effort. The developer must first choose the relevant parameters. Then, either manually write a predictor using these parameters and detailed knowledge of the computation or build a regression, which requires gathering execution data for several inputs. The parameters are easy to choose in some applications. For instance, they can be a matrix's width and height. However, in other applications, the decision is far more complicated. In a graph, for instance, the number of nodes and edges may be insufficient to predict the computational cost of some computation if it depends, for example, in how the edges are distributed among the nodes. Depending on the case this information may not be readily available at runtime. Furthermore, the interactions between the threads, due to memory contention or a shared cache level, for example, can differ between the training phase and the real execution of the application.

Both of these performance models require the gathering of timing data for each task version, possibly several times for different inputs. This procedure must be repeated if, for example, a new accelerator is installed. If the system is shared among untrusted users, the timings must be updated whenever some mitigation to some hardware vulnerability is installed if it affects performance, as is the case with Meltdown and Spectre for example. Moreover, when the performance depends strongly on the data, as is the case of many operations on a sparse matrix or graph, it may be difficult to obtain good predictors with these methods.

Instead of using a regression or requiring the developer to provide the runtime with a performance predictor, StarPU can measure the performance of the different task versions at runtime by using the History-based model. These measurements can be saved and used in subsequent executions of the application. The runtime executes each task version some user-defined number of times, whose default is 10. After this number of execution is reached, it predicts the execution time as the average time of the past executions.

5.3 Concluding Remarks

Both OmpSs and StarPU face the same challenge: they must not only decide when a task should run, and which computing unit should execute it, but also which of the multiple task versions to use. When only using data gathered during the execution, which is transparent to the programmer, both runtimes approach this problem the same way. Both compute the average execution time of each task version independently for the different data sizes. Both compute the average in the same way, with the use of the arithmetic mean instead of, for instance, giving more weight to more recent executions. Our Mean policy takes this same approach. While it is a simple policy, it leaves the performance at the mercy of the number of times each task version is repeated before the policy commits to the fastest version for that device. If this number is relatively high compared to the total number of tasks the application creates, which the runtime does not generally know, and the number of different versions, the policy spends too much time exploring, to the detriment of making use of the knowledge. If, however, the number is relatively low, the policies risk running a suboptimal version during the exploitation phase and obtaining worse performance than they could have otherwise.

While this approach works when the differences between the versions are large, this class of strategy, with distinct exploration and exploitation phases, is known to be necessarily suboptimal [GLK16b]. Other predictors, like UCB, can overcome this limitation. However, as we have shown in Chapter 4, depending on the computation, the simpler policy of taking the mean can be sufficient.

6 Conclusion and Future Work

This chapter is divided in two parts. In Section 6.1 we present the main contributions of this thesis. In Section 6.2, we describe some open issues and discuss directions which may be of interest for future works in this subject.

6.1 Contributions

We introduce two task version selection policies based on two algorithms for the multi-armed bandit problem: Upper Confidence Bound and Gradient Bandit. We also use a policy based on the average execution time of the different versions, Mean, which uses an Explore-Then-Commit strategy. This last policy is equivalent to the version selection part of the multi-version-aware task schedulers of StarPU and OmpSs: it executes each version some previously-defined number of times and, then, always uses the version which performed the best on average. The GB policy executes task versions randomly. As the application executes, the policy adjusts the probability of each version being selected according to their past performance. As a version outperforms the others, its probability of being chosen to execute increases. In the same way, if a version performs poorly, the policy will decrease that version's odds of being chosen for execution. However, even poorly-performing versions still have a chance of executing, so the policy is guaranteed to eventually identify the best-performing version if a sufficiently high number of tasks is created. The UCB policy does not make decisions randomly, but, like the Mean policy, it always uses the version it believes to be the best. It differs from the Mean policy by, instead of directly using the average performance of a version, building a confidence bound for each version, such that it knows, with some arbitrary level of confidence, how fast each version is likely to be.

We compare these three policies in three scenarios: a Cholesky factorization, with two vastly different versions for each of three of its linear algebra kernels; a matrix multiplication, with varying degrees of difference between hundreds of task versions generated through autotuning; and a matrix multiplication, with a few task versions with similar performance, also autotuned but manually chosen. In many experiments the GB policy failed to obtain reasonable performance, continuing to use inefficient versions frequently during the whole execution of the applications.

The Mean policy succeeded in identifying the right version to use. However, the performance of the policy depends on correctly deciding how many times each version should be executed before the policy commits. By committing too soon, the policy is liable to commit to an inefficient version. By committing too late, the policy risks not having enough time to take advantage of the acquired knowledge. The UCB policy also managed to identify which version it should use to improve performance. However, it does not always perform as well as the Mean policy with a good number of repetitions before commitment. Despite that, the UCB policy is also less susceptible to perform as badly as the Mean policy does when its number of repetitions is chosen poorly. This makes the UCB policy a feasible alternative, especially in situations where there are many different types of tasks with varied numbers of versions each.

Lastly, we add multi-versioned task support to OpenMP. We extend the Clang compiler to support our additions to the "omp task" pragma. And we extend the LLVM OpenMP runtime to be able to take advantage of these multiple versions, either using a policy similar to some of the policies of StarPU and OmpSs or with the UCB and GB policies.

6.2 Open Issues and Future Work

Our analysis considers a limited range of parameters. While UCB uses, among others, a parameter value which provides some theoretical guarantees that other values do not provide, only three values are used for the parameter of the Mean policy. It is possible, for example, that some other number of repetitions works better than those found by us. Moreover, the GB policy, which in most of the cases we analyze does not perform well, also only used three different parameter values. Likewise, it is possible it would have performed better with some other value. This, however, illustrates the programming cost associated with a high dependency on parameter values.

Other runtime systems make use of Explore-Then-Commit policies for choosing a task version. It would be interesting to use an algorithm like Upper Confidence Bound instead and see how it affects not only the online policies but also the training when using measurements collected offline, and if the behaviour differs in a heterogeneous environment as opposed to our homogeneous experiments.

Our approach suffers from some limitations. While we extend OpenMP tasks, other OpenMP constructs, like parallel loops and task loops, could also benefit from multiple versions. However, for purely technical reasons, our extension only

supports tasks. Constructs such as parallel loops can be of interest for a multi-versioned approach since the threads in the team will often execute the same code in different iterations of the loop, which may allow a multi-versioned policy to better tell how each version affects the others and reduce measurement noise from other tasks.

We have a single UCB policy, that is, we provide only one way of calculating its confidence bounds. However, there exist several variations of UCB. While the algorithm is usually the same, in some cases with different ways to guarantee versions with bad bounds are still executed from time to time, there are multiple ways to compute the confidence bounds.

In our OpenMP implementation, when a task finishes execution, its timing data is shared between all workers. This is done by updating a central structure. This can help workers gather information quicker than if they were working alone. However, as updates of the same task version, by different threads, must be synchronized, it also limits scalability. One option to increase scalability would be to keep statistics on a per-worker level, or perhaps per socket. Splitting these statistics would also enable the use of the multi-versioned OpenMP runtime with a heterogeneous architecture like the ARM big.LITTLE, since the two core types have very different performance profiles.

Bibliography

[ACF02] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.2 (May 2002), pp. 235–256 (cit. on pp. 43, 44).

[Acu+14] Bilge Acun, Abhishek Gupta, Nikhil Jain, et al. "Parallel Programming with Migratable Objects: Charm++ in Practice". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 647–658 (cit. on p. 2).

[Agr95] Rajeev Agrawal. "Sample mean based index policies by O(log n) regret for the multi-armed bandit problem". In: *Advances in Applied Probability* 27.4 (1995), pp. 1054–1078 (cit. on p. 43).

[And+90] E. Anderson, Z. Bai, J. Dongarra, et al. "LAPACK: A Portable Linear Algebra Library for High-performance Computers". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. New York, New York, USA: IEEE Computer Society Press, 1990, pp. 2–11 (cit. on p. 65).

[Ans+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, et al. "OpenTuner: An Extensible Framework for Program Autotuning". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT '14. Edmonton, AB, Canada: ACM, 2014, pp. 303–316 (cit. on pp. 25, 30).

[ATN10] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures". In: *Euro-Par 2009 – Parallel Processing Workshops*. Ed. by Hai-Xiang Lin, Michael Alexander, Martti Forsell, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 56–65 (cit. on p. 93).

[Aue+95] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. "Gambling in a rigged casino: The adversarial multi-armed bandit problem". In: *Proceedings of IEEE 36th Annual Foundations of Computer Science*. Oct. 1995, pp. 322–331 (cit. on p. 42).

[Aug+11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1631 (cit. on pp. 2, 15, 37).

[Aum+18] Olivier Aumage, François Broquedis, Pierrick Brunet, et al. *KSTAR OpenMP Compiler*. http://kstar.gforge.inria.fr. Accessed: 2018-03-28. 2018 (cit. on p. 15).

[Bal+13] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20 (cit. on p. 63).

[Bar+11] G. Barbato, E. M. Barini, G. Genta, and R. Levi. "Features and performance of some outlier detection methods". In: *Journal of Applied Statistics* 38.10 (2011), pp. 2133–2149. eprint: https://doi.org/10.1080/02664763.2010.545119 (cit. on p. 19).

[BCI19] Université de Bordeaux, CNRS, and Inria. *StarPU Handbook*. http://starpu.gforge.inria.fr/doc/html/index.html. Accessed: 2018-03-28. 2019 (cit. on pp. 16, 91).

[Ben+14] Siegfried Benkner, Franz Franchetti, Hans Michael Gerndt, and Jeffrey K. Hollingsworth. "Automatic Application Tuning for HPC Architectures (Dagstuhl Seminar 13401)". In: *Dagstuhl Reports* 3.9 (2014). Ed. by Siegfried Benkner, Franz Franchetti, Hans Michael Gerndt, and Jeffrey K. Hollingsworth, pp. 214–244 (cit. on p. 24).

[Big+18a] Bigot, Julien, Grandgirard, Virginie, Latu, Guillaume, et al. "Building and Auto-Tuning Computing Kernels: Experimenting with Boast and Starpu in the Gysela Code". In: *ESAIM: ProcS* 63 (2018), pp. 152–178 (cit. on p. 24).

[Big+18b] Julien Bigot, Virginie Grandgirard, Guillaume Latu, et al. "Building and Auto-Tuning Computing Kernels: Experimenting with BOAST and StarPU in the GYSELA Code". In: *ESAIM: Proceedings*. CEMRACS 2016 - Numerical challenges in parallel scientific computing 63 (2018) (June 2018), pp. 152–178 (cit. on p. 24).

[Bil97] J. Bilmes. "Optimizing Matrix Multiply using PHiPAC : A Portable High Performance, ANSI C Coding Methodology". In: *Proc. ICS 97* (1997) (cit. on p. 2).

[Blu+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, et al. "Cilk: An Efficient Multithreaded Runtime System". In: *Journal of Parallel and Distributed Computing* 37.1 (1996), pp. 55–69 (cit. on p. 2).

[Boa18] OpenMP Architecture Review Board. *Openmp 5.0 specification*. https://www.openmp.org/wp-content/uploads/OpenMP-APISpecification-5.0.pdf. Accessed: 2018-03-13. 2018 (cit. on pp. 2, 10).

[Bro+10] D Brown, Paul Messina, D Keyes, et al. "Scientific grand challenges: Crosscutting technologies for computing at the exascale". In: *Office of Science, US Department of Energy, February* (2010), pp. 2–4 (cit. on p. 2).

[Cen18] Barcelona Supercomputing Center. *OmpSs Documentation*. https://pm.bsc.es/ftp/ompss/doc/examples/. Accessed: 2018-03-28. 2018 (cit. on p. 14).

[CH07] P. Conway and B. Hughes. "The AMD Opteron Northbridge Architecture". In: *IEEE Micro* 27.2 (Mar. 2007), pp. 10–21 (cit. on p. 7).

[Cha+09] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. "Evaluation of the Intel® Core™ i7 Turbo Boost feature". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Oct. 2009, pp. 188–197 (cit. on p. 38).

[DD51] R. B. Dean and W. J. Dixon. "Simplified Statistics for Small Numbers of Observations". In: *Analytical Chemistry* 23.4 (1951), pp. 636–638. eprint: https://doi.org/10.1021/ac60052a025 (cit. on p. 19).

[Din+15] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, et al. "Autotuning Algorithmic Choice for Input Sensitivity". In: *SIGPLAN Not.* 50.6 (June 2015), pp. 379–390 (cit. on pp. 34, 35).

[Don19] J. Dongarra. *Top500: TOP 500 Supercomputer Sites*. https://www.top500.org. 2019 (cit. on p. 1).

[Dup+08] F. Dupros, H. Aochi, A. Ducellier, D. Komatitsch, and J. Roman. "Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation". In: *2008 11th IEEE International Conference on Computational Science and Engineering*. July 2008, pp. 253–260 (cit. on p. 33).

[Dup+09] Fabrice Dupros, Christiane Pousa Ribeiro, Alexandre Carissimi, and Jean-Francois Mehaut. "Parallel Seismic Wave Propagation on NUMA architectures". In: *Proceedings of International Conference on Parallel Computing (ParCO)*. Vol. 19. Advances in Parallel Computing. Lyon, France, 2009, pp. 67–74 (cit. on p. 33).

[Dur+11] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, et al. "OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures". In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193. eprint: https://doi.org/10.1142/S0129626411000151 (cit. on pp. 2, 13, 37).

[FJ98] M. Frigo and S. G. Johnson. "FFTW: an adaptive software architecture for the FFT". In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*. Vol. 3. May 1998, 1381–1384 vol.3 (cit. on pp. 2, 24).

[Fly66] M. J. Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (Dec. 1966), pp. 1901–1909 (cit. on p. 6).

[FPP07] David Freedman, Robert Pisani, and Roger Purves. *Statistics (4th edn)*. 2007 (cit. on pp. 20, 21).

[Gaj+14] Vladimir Gajinov, Srđan Stipić, Igor Erić, et al. "DaSH: A Benchmark Suite for Hybrid Dataflow and Shared Memory Programming Models: with Comparative Evaluation of Three Hybrid Dataflow Models". In: *Proceedings of the 11th ACM Conference on Computing Frontiers*. CF '14. Cagliari, Italy: ACM, 2014, 4:1–4:11 (cit. on pp. 65, 85).

[Gal95] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1995 (cit. on p. 9).

[GLK16a] Aurelien Garivier, Tor Lattimore, and Emilie Kaufmann. "On Explore-Then-Commit strategies". In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett. Curran Associates, Inc., 2016, pp. 784–792 (cit. on p. 44).

[GLK16b] Aurélien Garivier, Tor Lattimore, and Emilie Kaufmann. "On explore-then-commit strategies". In: *Advances in Neural Information Processing Systems*. 2016, pp. 784–792 (cit. on pp. 39, 41, 94).

[Gru69] Frank E. Grubbs. "Procedures for Detecting Outlying Observations in Samples". In: *Technometrics* 11.1 (1969), pp. 1–21. eprint: https://www.tandfonline.com/doi/pdf/10.1080/00401706.1969.10490657 (cit. on p. 19).

[GS16] Unai Garciarena and Roberto Santana. “Evolutionary Optimization of Compiler Flag Selection by Learning and Exploiting Flags Interactions”. In: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*. GECCO ’16 Companion. Denver, Colorado, USA: ACM, 2016, pp. 1159–1166 (cit. on p. 35).

[Hag+16] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. “Exploring performance and power properties of modern multi-core chips via simple machine models”. In: *Concurrency and Computation: Practice and Experience* 28.2 (2016), pp. 189–210. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3180 (cit. on p. 1).

[HE08] Kenneth Hoste and Lieven Eeckhout. “Cole: Compiler Optimization Level Exploration”. In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’08. Boston, MA, USA: ACM, 2008, pp. 165–174 (cit. on p. 35).

[Hon] Hoon Hong. “Parallel Symbolic Computation PASCO ’94”. In: *Parallel Symbolic Computation PASCO ’94*, pp. 1–448. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789814533584 (cit. on p. 9).

[Hor+13] M. Horowitz, F. Labonte, O. Shacham, et al. *A Journey to Exascale Computing*. https://science.energy.gov/~/media/ascr/ascac/pdf/reports/2013/SC12_Harrod.pdf. Accessed: 2018-03-12. 2013 (cit. on p. 6).

[Jar30] Vojtech Jarnık. “About a certain minimal problem”. In: *Práce Moravské Prırodovedecké Spolecnosti* 6 (1930), pp. 57–63 (cit. on p. 33).

[KHF18] Hassan N Khan, David A Hounshell, and Erica RH Fuchs. “Science and research policy at the end of Moore’s law”. In: *Nature Electronics* 1.1 (2018), p. 14 (cit. on p. 6).

[Kru56] Joseph B. Kruskal. “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem”. In: *Proceedings of the American Mathematical Society* 7.1 (1956), pp. 48–50 (cit. on p. 33).

[LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75– (cit. on pp. 36, 51).

[LAC] James Lagrone, Ayodunni Aribuki, and Barbara Chapman. *A Set of Microbenchmarks for Measuring OpenMP Task Overheads* (cit. on p. 35).

[Lai87] Tze Leung Lai. “Adaptive Treatment Allocation and the Multi-Armed Bandit Problem”. In: *Ann. Statist.* 15.3 (Sept. 1987), pp. 1091–1114 (cit. on p. 44).

[LGP05] Xiaoming Li, Maria Jesus Garzaran, and David Padua. “Optimizing Sorting with Genetic Algorithms”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 99–110 (cit. on p. 35).

[Lib15] GNU Libgomp. *GNU Offloading and Multi Processing Runtime Library: The GNU OpenMP and OpenACC Implementation*. Tech. rep. 2015 (cit. on p. 12).

[Lim+17] J. V. F. Lima, I. Raïs, L. Lefevre, and T. Gautier. "Performance and Energy Analysis of OpenMP Runtime Systems with Dense Linear Algebra Algorithms". In: *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. Oct. 2017, pp. 7–12 (cit. on p. 12).

[LLS08] Lixia Liu, Zhiyuan Li, and Ahmed H. Sameh. "Analyzing Memory Access Intensity in Parallel Programs on Multicore". In: *Proceedings of the 22Nd Annual International Conference on Supercomputing*. ICS '08. Island of Kos, Greece: ACM, 2008, pp. 359–367 (cit. on p. 1).

[LS18] Tor Lattimore and Csaba Szepesvári. "Bandit algorithms". In: *preprint* (2018) (cit. on p. 30).

[Luc+14] Robert Lucas, James Ang, Keren Bergman, et al. "Top ten exascale research challenges". In: *Office of Science, US Department Of Energy* (2014) (cit. on p. 2).

[Mat+13] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. "System V application binary interface". In: *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013) (cit. on p. 57).

[MM17] Luis Felipe Millani and Lucas Mello Schnorr. "Computation-Aware Dynamic Frequency Scaling: Parsimonious Evaluation of the Time-Energy Trade-Off Using Design of Experiments". In: *Euro-Par 2016: Parallel Processing Workshops*. Ed. by Frédéric Desprez, Pierre-François Dutot, Christos Kaklamanis, et al. Cham: Springer International Publishing, 2017, pp. 583–595 (cit. on p. 5).

[Moo+65] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965 (cit. on p. 6).

[MRT12] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012 (cit. on p. 22).

[NMC15] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. "Use of Previously Acquired Positioning of Optimizations for Phase Ordering Exploration". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '15. Sankt Goar, Germany: ACM, 2015, pp. 58–67 (cit. on p. 35).

[Pei52] B. Peirce. "Criterion for the rejection of doubtful observations". In: *The Astronomical Journal* 2 (July 1852), pp. 161–163 (cit. on p. 20).

[Pla+13] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. "Self-Adaptive OmpSs Tasks in Heterogeneous Environments". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. May 2013, pp. 138–149 (cit. on p. 91).

[Pri57] R. C. Prim. "Shortest connection networks and some generalizations". In: *The Bell System Technical Journal* 36.6 (Nov. 1957), pp. 1389–1401 (cit. on p. 33).

[Pus+05] M. Puschel, J. M. F. Moura, J. R. Johnson, et al. "SPIRAL: Code Generation for DSP Transforms". In: *Proceedings of the IEEE* 93.2 (Feb. 2005), pp. 232–275 (cit. on p. 24).

[Rah+16] Amir M Rahmani, Pasi Liljeberg, Ahmed Hemani, Axel Jantsch, and Hannu Tenhunen. *The Dark Side of Silicon*. Springer, 2016 (cit. on p. 1).

[Rei07] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007 (cit. on p. 2).

[Rob52] Herbert Robbins. “Some aspects of the sequential design of experiments”. In: *Bulletin of the American Mathematical Society* 58.5 (1952), pp. 527–535 (cit. on p. 30).

[Rup18] Karl Rupp. *42 Years of Microprocessor Trend Data*. https://github.com/karlrupp/microprocessor-trend-data. Accessed: 2018-03-12. 2018 (cit. on p. 6).

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018 (cit. on pp. 23, 40, 44, 45).

[Sed78] Robert Sedgewick. “Implementing Quicksort Programs”. In: *Commun. ACM* 21.10 (Oct. 1978), pp. 847–857 (cit. on p. 35).

[SQP08] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. “Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs”. In: *SIGARCH Comput. Archit. News* 36.1 (Mar. 2008), pp. 277–286 (cit. on p. 1).

[SRD16] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. “Matrix Multiplication Beyond Auto-tuning: Rewrite-based GPU Code Generation”. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES ’16. Pittsburgh, Pennsylvania: ACM, 2016, 15:1–15:10 (cit. on p. 35).

[Tri+03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. “Compiler Optimization-space Exploration”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’03. San Francisco, California, USA: IEEE Computer Society, 2003, pp. 204–215 (cit. on p. 34).

[Tuk77] John W. Tukey. “Exploratory data analysis”. In: *Addison-Wesley series in behavioral science : quantitative methods*. 1977 (cit. on p. 19).

[TW17] Thomas N. Theis and H.-S. Philip Wong. “The End of Moore’s Law: A New Beginning for Information Technology”. In: *Computing in Science & Engineering* 19.2 (2017), pp. 41–50. eprint: https://aip.scitation.org/doi/pdf/10.1109/MCSE.2017.29 (cit. on p. 1).

[VD00] Richard Vuduc and James W. Demmel. “Code Generators for Automatic Tuning of Numerical Kernels: Experiences with FFTW Position Paper”. In: *Semantics, Applications, and Implementation of Program Generation*. Ed. by Walid Taha. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 190–211 (cit. on p. 24).

[Ven+10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, et al. “Conservation Cores: Reducing the Energy of Mature Computations”. In: *SIGARCH Comput. Archit. News* 38.1 (Mar. 2010), pp. 205–218 (cit. on p. 1).

[Vid+18] Brice Videau, Kevin Pouget, Luigi Genovese, et al. “BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications”. In: *The International Journal of High Performance Computing Applications* 32.1 (2018), pp. 28–44. eprint: https://doi.org/10.1177/1094342017718068 (cit. on p. 30).

[Wan+13] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. "AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs". In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2013, pp. 1–12 (cit. on p. 66).

[Wan+14] Endong Wang, Qing Zhang, Bo Shen, et al. "Intel Math Kernel Library". In: *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*. Cham: Springer International Publishing, 2014, pp. 167–188 (cit. on pp. 24, 66).

[Wel62] B. P. Welford. "Note on a Method for Calculating Corrected Sums of Squares and Products". In: *Technometrics* 4.3 (1962), pp. 419–420. eprint: https://amstat.tandfonline.com/doi/pdf/10.1080/00401706.1962.10490022 (cit. on p. 59).

[WM95] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24 (cit. on p. 7).

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. "Automated empirical optimizations of software and the ATLAS project". In: *Parallel Computing* 27.1 (2001). New Trends in High Performance Computing, pp. 3–35 (cit. on pp. 2, 24).

[Xio+01] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. "SPL: A Language and Compiler for DSP Algorithms". In: *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*. PLDI '01. Snowbird, Utah, USA: ACM, 2001, pp. 298–308 (cit. on p. 2).

[Zha+08] Li Zhao, Ravi Iyer, Mike Upton, and Don Newell. "Towards Hybrid Last Level Caches for Chip-multiprocessors". In: *SIGARCH Comput. Archit. News* 36.2 (May 2008), pp. 56–63 (cit. on p. 7).

[Zhu+13] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. "Survey of Energy-Cognizant Scheduling Techniques". In: *IEEE Transactions on Parallel and Distributed Systems* 24.7 (July 2013), pp. 1447–1464 (cit. on p. 5).

[Zia+10] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek. "Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures". In: *2010 18th IEEE Symposium on High Performance Interconnects*. Aug. 2010, pp. 1–6 (cit. on p. 7).

List of Figures

List of Tables

A Autotuning Example

Figure A.1 shows the effect on the execution time of changing the value of TILE_J. The vertical axis shows the execution time, in seconds. The horizontal axis shows the value used for TILE_J. Each point is the minimum execution time of all executions where the TILE_J has the value shown in the horizontal axis. Again, we can easily see TILE_J influences execution time. TILE_J has a much stronger effect on performance than TILE_I does, with the execution time varying between 1.8 second at 256 and 9.6 seconds at 1. Unlike with TILE_I, TILE_J does not looks likely to improve performance by much through further refiniments as the execution times with a value of 64 or more are very similar.

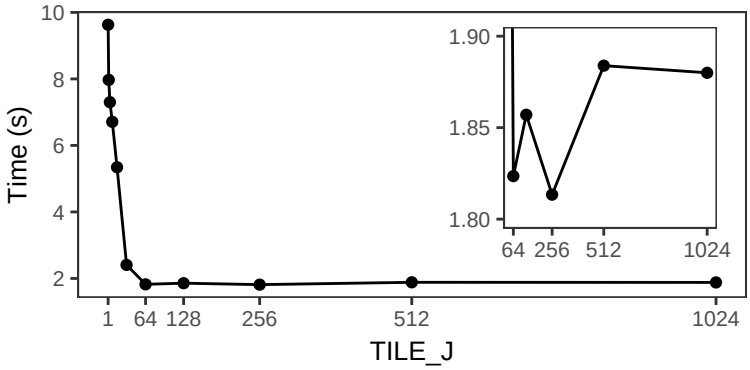

Figure A.1.: Effect of different values of TILE_J on the execution time. Each point, one for every value of TILE_J, represents the execution time using that value of TILE_J and the values of TILE_I and TILE_K which give the lowest execution time for that TILE_J.

Figure A.2 shows the effect on the execution time of changing the value of TILE_K. The vertical axis shows the execution time, in seconds. The horizontal axis shows the value used for TILE_K. Each point is the minimum execution time of all executions where the TILE_K has the value shown in the horizontal axis. Again, we can easily see TILE_K influences execution time, and like TILE_J the effect is high: the execution time can vary between 1.8 second at 1 and 10.5 seconds at 512. Like TILE_I, the curve between 4 and 32 indicates a local optimum. However, these values perform worse than 1 does so it is unlikely the local optimum will perform better.

Figure A.3 shows how changing the value of TILE_J affects performance. The differences between the execution times obtained with these values of TILE_J are

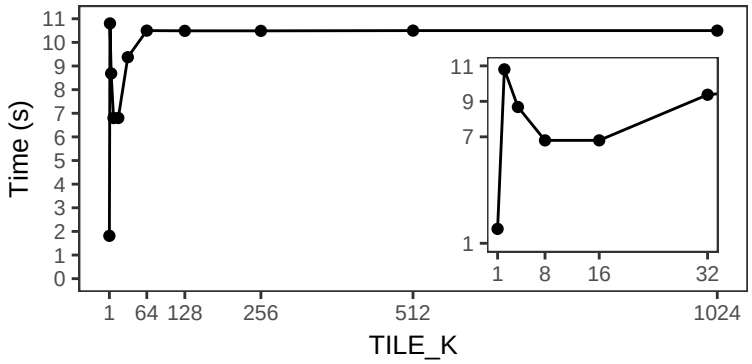

Figure A.2.: Effect of different values of TILE_K on the execution time. Each point, one for every value of TILE_K, represents the execution time using that value of TILE_K and the values of TILE_I and TILE_J which give the lowest execution time for that TILE_K.

much smaller than those for TILE_I. We can see the difference between 256, the best value found in the previous autotuning step, and 128, the best value found on this step, is of only 1.1%, as the point 128 took 1.79 second and 256 took 1.81 second. However, as the number of executions we have is very small, we cannot say there is an actual difference between these points. To find whether the difference is there, we repeat the experiment 50 times. These results are shown in Figure A.4. We can see that while 128 seemed to be better than 256, now that we have better precision on the results it seems 128 is actually worse than 256, and of those points it is actually 224 which improves execution time the most, improving the execution time by 1.4% compared to the previous iteration. Notice this is the same point mentioned at the end of the last paragraph.

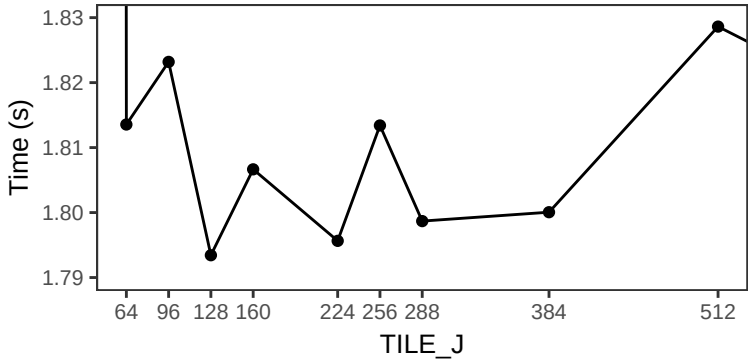

Figure A.3.: Effect of different values of TILE_J on the execution time. Each point, one for every value of TILE_J, represents the execution time using that value of TILE_J and the values of TILE_I and TILE_K which give the lowest execution time for that TILE_J.

Lastly, for TILE_K For the sake of completeness, the confidence intervals at 99.7% are shown in Figure A.6 and confirm the value of 1 is the one that should be used for TILE_K.

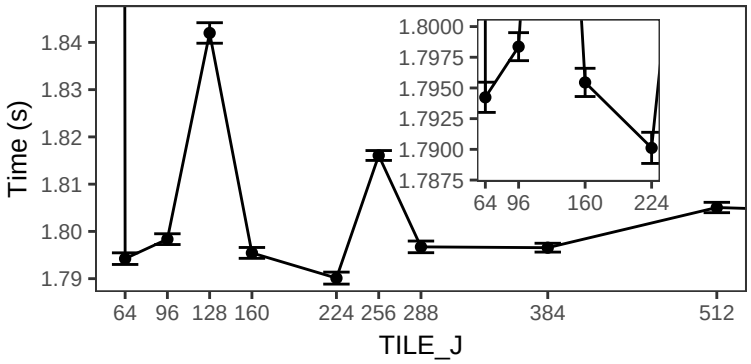

Figure A.4.: Effect of different values of TILE_J on the execution time. Each point, one for every value of TILE_J, represents the execution time using that value of TILE_J and the values of TILE_I and TILE_K which give the lowest execution time for that TILE_J. Each point is the average of 50 executions and the bars indicate its confidence interval at a 99.7% confidence level.

Lastly, Figure A.5 shows how changing the value of TILE_K affects performance, and confirms that this parameter should be set at 1. Figure A.6 shows the confidence intervals after repeating the experiment 50 times. As expected due to the large difference in execution times, there is a significative difference, at a 99.7% confidence level, between using 1 for this parameter or the second-best value found, 8.

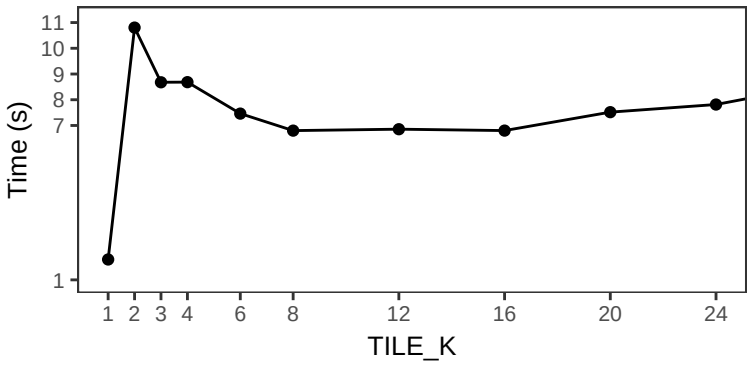

Figure A.5.: Effect of different values of TILE_K on the execution time. Each point, one for every value of TILE_K, represents the execution time using that value of TILE_K and the values of TILE_I and TILE_J which give the lowest execution time for that TILE_K.

I

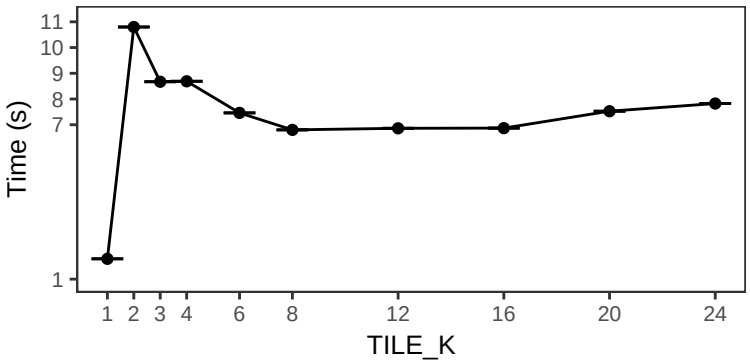

Figure A.6.: Effect of different values of TILE_K on the execution time. Each point, one for every value of TILE_K, represents the execution time using that value of TILE_K and the values of TILE_I and TILE_J which give the lowest execution time for that TILE_K. Each point is the average of 50 executions and the bars indicate its confidence interval at a 99.7% confidence level.

Cholesky Decomposition Trace

B

Figure B.1 shows which kernel version was used during the first 200 times each of the three kernels was executed when running the benchmark with a single thread and a 16384 by 16384 input matrix. The horizontal axis represents the logical time for each kernel. The vertical axis shows the different policies, UCB, Mean, and GB, with the parameter used, each corresponding to an independent execution of the benchmark. The colours show the kernel version used, with red for Intel MKL and blue for OpenBLAS. Lastly, the first, second, and third columns correspond to the *sgemm*, *ssyrk*, and *strsm* kernels, respectively. We can see in this case the UCB policy, shown at the top, regardless of the value of its parameter, did not execute the MKL version with the *sgemm* and *ssyrk* kernels except in the very beginning of the execution. With the *strsm* kernel, however, the MKL version is still run a few more times during the execution since the performance difference between MKL and OpenBLAS for this operation is smaller than for the other two operations. The Mean policy executes both versions the defined number of times, 30 in this case, and does not switch for the remaining of the execution. The GB policy with a large value for its parameter gave preference to the MKL version with the *sgemm* and *ssyrk* kernels, and to OpenBLAS with the *strsm* kernel. In both cases, however, it continued sometimes executing the other version later in the execution. When using a smaller parameter value, the policy showed less preference between the two versions. That is only in the beginning of execution, however. Figure B.2 shows the execution in its entirety instead of only its first 200 kernel executions, and we can see GB with a high value for its parameter eventually switches to OpenBLAS, while GB with a low value keeps switching between the two versions during the whole execution.

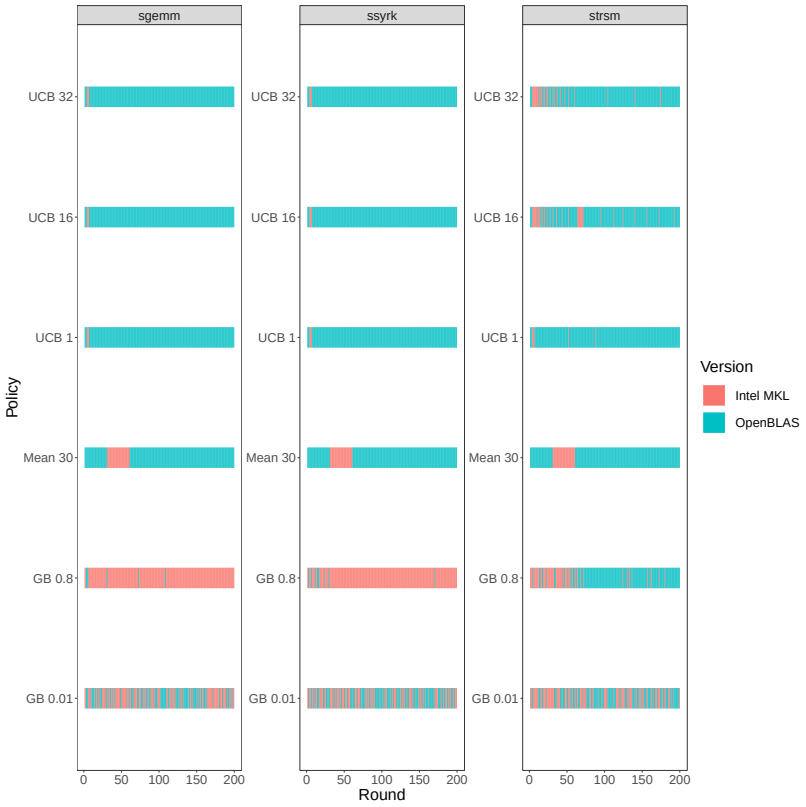

Figure B.1.: Choices made by the different policies, with different parameters, along the first 200 executions of the *sgemm*, *ssyrk*, and *strsm* kernels of the Cholesky benchmark.

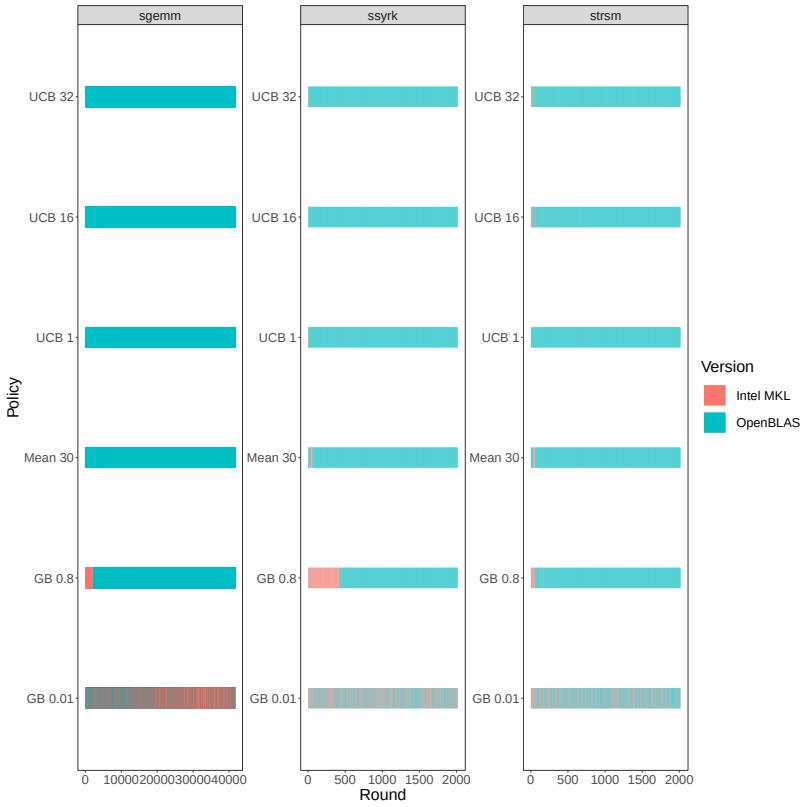

Figure B.2.: Choices made by the different policies, with different parameters, along the execution of the *sgemm*, *ssyrk*, and *strsm* kernels of the Cholesky benchmark.